ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Information and Computation 182 (2003) 14–52

Information and Computation

www.elsevier.com/locate/ic

Pair-independence and freeness analysis through linear refinement

Giorgio Levi[a] and Fausto Spoto[b,*]

[a] *Dipartimento di Informatica, Via F. Buonarroti 2, 56127 Pisa, Italy*
[b] *Dipartimento di Informatica, Strada Le Grazie, 15, 37134 Ca' Vignal, Verona, Italy*

Received 29 May 2001

Abstract

Linear refinement is a technique for systematically constructing more precise abstract domains for program analysis starting from the basic domain which represents just the property of interest. We use here linear refinement to construct a domain for pair-independence and freeness analysis of logic programs which is strictly more precise than Jacobs and Langen's domain for sharing analysis endowed with freeness information. Moreover, it can be used for abstract compilation, while Jacobs and Langen's domain can only be used for abstract interpretation. We provide an approximate representation of our domain and algorithms for the abstract operations. We describe an implementation of an analyser which uses abstract compilation over our domain and its evaluation over a set of benchmarks. This shows that its precision is comparable to that of a traditional sharing and freeness analysis performed through abstract interpretation. To the best of our knowledge, this is the first implementation of a sharing analysis based on abstract compilation, as well as the first implementation of a static analysis based on a new domain developed through linear refinement.

1. Introduction

This paper is concerned with the systematic design, by means of linear refinement, of a new abstract domain for two important properties of logic programs, i.e., pair-independence and freeness. Pair-independence analysis [4,38] is concerned with determining at compile-time a superset of the set of pairs of variables which, in a given program point, can be bound at run-time

* Corresponding author.
E-mail addresses: levi@di.unipi.it (G. Levi), spoto@sci.univr.it (F. Spoto).

0890-5401/03/$ - see front matter
doi:10.1016/S0890-5401(02)00048-2

to two terms which share some variable. It is a particular case of set-(in)dependence analysis, also called *sharing* analysis [6,7,28–31,36]. In set-independence analysis, not only pairs but sets of variables are considered. (In)dependence analysis is useful for avoiding occur-check [38] and for automatic program parallelisation [28,36]. As stressed in [4], pair-(in)dependence information is actually needed in program analysis and transformation, and set-(in)dependence information is redundant w.r.t. pair-(in)dependence information.

Freeness analysis [6,7,10,11,27,30,36] is concerned with determining at compile-time a subset of variables which are guaranteed to be bound at run-time to some variable in a given program point. Freeness analysis is useful for optimising unification, for goal reordering, for avoiding type checking and, again, in automatic program parallelisation. It is well known that performing sharing and freeness analysis in conjunction improves the precision of both [28,36].

Linear refinement [24] is a technique for systematically constructing abstract domains for program analysis. Given a basic abstract domain representing just the property of interest and a concrete operation (which, since we are considering logic programs, is usually unification) a new more accurate domain is constructed. The new domain leads to more precise abstract operations.

The first contribution of this paper is the definition, through abstract interpretation [16,17] and linear refinement, of a new domain for pair-independence and freeness. The use of linear refinement for the definition of our domain, differently from the $Sharing \times Free$ domain [28,36], leads to simple and general definitions and proofs. It is worth noting that the original correctness proof for the domain *Sharing* is very complex and uses a large part of Langen's PhD thesis [31]. We also show, *within the linear refinement framework*, why and how independence information interacts with freeness information. An important feature of our domain is that it can be used for abstract compilation [22,26], which is an application of abstract interpretation where, rather than computing the abstract denotation of a program by executing its *concrete* code over abstract data, the code itself is abstracted and replaced by *abstract code*, where concrete data structures are replaced by their abstraction. As a consequence, the computation of the abstract denotation can be achieved by the same algorithm used in the concrete computation.

The second contribution is the design of a computationally feasible representation of our domain, together with algorithms for computing an approximation of the concrete operations. Since this approximation can reduce the theoretical precision of our domain, we describe a prototypical analyser for pair-independence and freeness, based on abstract compilation and a fixpoint semantics. The use of a fixpoint semantics results in a goal-independent analysis. This means that the program is analysed for the most general goals only. More instantiated goals are analysed by using the analysis of the most general goals. We evaluate our analyser over a set of benchmarks. Although it is just a prototype, our evaluation shows that it is efficient enough for practical use on small benchmarks. Its precision is shown to be comparable to that of a traditional goal-dependent analysis.

To the best of our knowledge, this is the first implementation of a goal-independent sharing analysis based on abstract compilation, as well as the first implementation of a static analysis based on a new domain developed through linear refinement.

This paper is organised as follows. Section 2 discusses related works. Section 3 introduces preliminary definitions. Sections 4 and 5 introduce two basic domains for pair-independence and freeness analysis, respectively, and show that their linear refinement does not lead to useful domains. In Section 6 we justify this result and in Section 7 we show why it is useful to combine the

two analyses by using a domain which is defined as the linear refinement of the reduced product of the basic domains for pair-independence and for freeness. This domain is shown to be more precise than the domain of [28,36]. Section 8 defines a data structure which can be used as an approximate representation for our new domain, together with algorithms for the abstract operations. Section 9 shows an algorithm for computing the abstraction map. Section 10 describes the implementation of a prototypical analyser, and Section 11 reports its evaluation over a set of benchmarks. Finally, Section 12 draws some conclusions. Most of the proofs are kept in a separate appendix, for the convenience of the reader.

Preliminary and partial versions of this paper appeared in [1,33].

2. Related work

Almost all the domains developed for sharing analysis are not amenable to abstract compilation [6,28–31,36]. Moreover, they have been developed without using any systematic technique like linear refinement.

To the best of our knowledge, only [7,13] provide abstract domains for sharing analysis which can be used for abstract compilation. The domain in [13] is isomorphic to the *Sharing* domain of [28,31]. This means that, when used for abstract compilation, in order to obtain a useful precision, it *must* be coupled with a domain expressing further information, like freeness or linearity. This domain must in turn be amenable to abstract compilation. We do not know of any prototypical analyser implemented through their domain. The domain in [7] models sharing, freeness and groundness, but it is *not* developed through abstract interpretation. Instead, it uses *pre-interpretations*.

In the context of logic languages, linear refinement has been already used for reconstructing the domain *Pos* for groundness analysis [37]. Moreover, it has been used to develop new domains for type [32] and freeness analysis [27].

3. Preliminaries

3.1. Terms, substitutions, and Herbrand constraints

We denote by $\wp(S)$ the *powerset* of a set S, by $\#S$ its *cardinality* and by $\wp_f(S)$ the set of all subsets of S of finite cardinality.

In this paper, we assume that $\mathcal{V}$ is an infinite set of *variables*, $V \in \wp_f(\mathcal{V})$ and Σ is a set of *function symbols* with associated *arity*, containing at least a symbol of arity 0. We define $terms(\Sigma, V)$ as the minimal set of terms built from V and Σ as: $V \subseteq terms(\Sigma, V)$ and if $t_1, \ldots, t_n \in terms(\Sigma, V)$ and $\mathtt{f} \in \Sigma$ has arity $n \geqslant 0$, then $\mathtt{f}(t_1, \ldots, t_n) \in terms(\Sigma, V)$. Let $t \in terms(\Sigma, V)$. By $vars(t)$ we denote the set of variables which *occur* in t. If $vars(t) = \varnothing$, then t is *ground*. It is *linear* if every $v \in V$ occurs at most once in t. If $x \in \mathcal{V}$ and then $V \cup x$ means $V \cup \{x\}$ and $V \setminus x$ means $V \setminus \{x\}$. *Syntactical substitution* in t of x with $t' \in terms(\Sigma, V)$ is denoted by $t[x \mapsto t']$.

A *substitution* θ is a map from variables to terms. Its *domain* is denoted by $dom(\theta)$ and the set of variables in its *range* by $rng(\theta)$. The set of *idempotent substitutions* θ such that $dom(\theta) \cup rng(\theta) \subseteq V$ and $dom(\theta) \cap rng(\theta) = \varnothing$ is denoted by Θ_V. We write $\theta \in \Theta_V$ extensionally as $\theta = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$, meaning that $dom(\theta) = \{v_1, \ldots, v_n\}$ and $\theta(v_i) = t_i$ for every $i = 1, \ldots, n$. Let $\theta \in \Theta_V$ and $R \subseteq V$. We define $\theta|_R(x) = \theta(x)$ if $x \in R$ and $\theta|_R(x) = x$ if $x \in V \setminus R$. If $t \in terms(\Sigma, V)$ then $t\theta \in terms(\Sigma, V)$ is the term obtained by replacing every variable x in t by $\theta(x)$. *Composition of substitutions* $\theta, \sigma \in \Theta_V$ is defined as $(\theta\sigma)(x) = \theta(x)\sigma$ for every $x \in V$. We recall that it is associative, the empty substitution ε is the neutral element and, for each term t, we have $t(\theta\sigma) = (t\theta)\sigma$.

The set C_V of finite sets of *Herbrand equations* is

$$C_V = \wp_f(\{t^1 = t^2 \,|\, t^1, t^2 \in terms(\Sigma, V)\}).$$

Every substitution can be seen as a set of Herbrand equations. The embedding map is $Eq(\theta) = \{v = \theta(v) \,|\, v \in dom(\theta)\}$. We hence assume that $\Theta_V \subseteq C_V$. Let $c \in C_V$. We say that $c\theta$ is *true* if $t^1\theta$ is syntactically equal to $t^2\theta$ for every $(t^1 = t^2) \in c$. We know [35] that if there exists $\theta \in \Theta_V$ such that $c\theta$ is true, then c can be put in the *normal form* $mgu(c) \in \Theta_V$ which is such that $c\theta$ is true if and only if $mgu(c)\theta$ is true. If no $\theta \in \Theta_V$ exists such that $c\theta$ is true, then $mgu(c)$ is undefined. Note that $c \in C_V$ in normal form can be seen as a substitution, and hence the notations $c(x)$ and tc are defined.

Let $\mathcal{W}$ be an infinite set of variables disjoint from $\mathcal{V}$. We define the set

$$H_V = \left\{ \exists_W c \,\middle|\, \begin{array}{l} W \in \wp_f(\mathcal{W}),\ c \in C_{V \cup W} \text{ and there exists} \\ \theta \in \Theta_{V \cup W} \text{ s.t. } rng(\theta) \subseteq V \text{ and } c\theta \text{ is true} \end{array} \right\}$$

of *existential Herbrand constraints*. Here, $\mathcal{V}$ are called the *program variables* and $\mathcal{W}$ the *existential variables*. Existential variables are the unnamed variables of Prolog. For instance, the most general solution of the following Prolog clause:

$$\mathtt{p(X)} : -\mathtt{X} = \mathtt{f(Y)}$$

is the existential Herbrand constraint $\exists_{\{Y\}}\{X = \mathtt{f}(Y)\}$.

We define

$$sol_V(\exists_W c) = \{\theta|_V \,|\, \theta \in \Theta_{V \cup W},\ rng(\theta) \subseteq V \text{ and } c\theta \text{ is true}\}.$$

Hence $sol_V(\exists_W c) = sol_V(\exists_W mgu(c))$. For instance, if $V = \{X, Z\}$, then $\{\{X = \mathtt{f}(a)\}, \{X = \mathtt{f}(\mathtt{f}(a))\}, \{X = \mathtt{f}(Z)\}, \{X = \mathtt{f}(\mathtt{f}(Z))\}\} \subseteq sol_V(\exists_{\{Y\}}\{X = \mathtt{f}(Y)\})$.

A constraint $\exists_W c$ is in *normal form* if c is in normal form. It is *consistent* if $sol_V(\exists_W c) \neq \varnothing$. Two constraints $h_1, h_2 \in H_V$ are *equivalent* if $sol_V(h_1) = sol_V(h_2)$. For instance, the constraints $\exists_{\{Y\}}\{X = \mathtt{f}(Y)\}$ and $\exists_{\{Z\}}\{X = \mathtt{f}(Z)\}$ are equivalent. In the following, a constraint will stand for its equivalence class. Since, as shown above, every consistent existential Herbrand constraint has an equivalent normal form, in the following we will consider only normal existential Herbrand constraints.

3.2. The s-semantics

We use H_V as the *computational domain* of *programs*. Since we will later define abstractions of H_V (Section 8), we decorate the following definitions with H_V. Once an abstraction will be defined, we just substitute it instead of H_V.

Definition 1. Let Π be a finite set of *predicate symbols* with associated arity. A *logic program* over H is a finite set of *clauses*

$$\mathtt{p}(X_1,\ldots,X_n) : -G_1,\ldots,G_m, \tag{1}$$

where $\mathtt{p}^n \in \Pi$ with $n \geqslant 0$, $\{X_1,\ldots,X_n\} \subseteq V$ are distinct and for every $i = 1,\ldots,m$ we have $G_i \in H_V$ or $G_i = \mathtt{q}(Y_1,\ldots,Y_l)$ with $\mathtt{q}^l \in \Pi$ and $\{Y_1,\ldots,Y_l\} \subseteq V$ distinct. The left-hand side of (1) is the *head* of the clause, the right-hand side is its *tail*. We say that the clause (1) *defines* the predicate $\mathtt{p}$. Every predicate must be defined by at least one clause of P. If more clauses of P define the same predicate, they must use the same variables $X_1,\ldots,X_n$ in (1).

The *s-semantics* of logic programs [5] is based on a fixpoint definition over *interpretations*. Interpretations work over the *collecting version* [17] of H_V, i.e., over the lattice $\langle \wp(H_V), \cap, \cup, H_V, \varnothing \rangle$.

Definition 2. An interpretation over H is a function I which maps every $\mathtt{p} \in \Pi$ to $\wp(H_{\{X_1,\ldots,X_n\}})$, where $\{X_1,\ldots,X_n\}$ are the variables in the head of the clauses which define $\mathtt{p}$ (Definition 1). The set of interpretations over H is denoted by $\mathbb{I}^H$.

Four operations over H_V, called *conjunction*, *restriction*, *expansion*, and *renaming*, respectively, are used to define the *s*-semantics. They are defined in Definition 3. The operation $\bigstar^{H_V}$ computes the conjunction of two constraints through the normalisation procedure. The *restrict* and *expand* operations remove a variable from and add a variable to a constraint, respectively. Note that *expand* is not the identity function but an embedding, as its signature shows. The operation *rename* gives a new name to a variable.

Definition 3. We define

$$\bigstar^{H_V} : H_V \times H_V \mapsto H_V,$$

$$\mathit{restrict}_x^{H_V} : H_V \mapsto H_{V\setminus x} \quad \text{with } x \in V,$$

$$\mathit{expand}_x^{H_V} : H_V \mapsto H_{V\cup x} \quad \text{with } x \notin V,$$

$$\mathit{rename}_{x\to n}^{H_V} : H_V \mapsto H_{(V\setminus x)\cup n} \quad \text{with } x \in V \text{ and } n \notin V$$

as[1]

$$(\exists_{W_1} c_1) \bigstar^{H_V} (\exists_{W_2} c_2) = \begin{cases} \exists_{W_1\cup W_2} \mathit{mgu}(c_1 \cup c_2) & \text{if } \mathit{mgu}(c_1 \cup c_2) \mathit{exists}, \\ \text{undefined} & \text{otherwise}, \end{cases}$$

$$\mathit{restrict}_n^{H_V}(\exists_W c) = \exists_{W\cup N} c[n \mapsto N] \quad \text{with } N \in \mathcal{W} \setminus W,$$

[1] In the definition of $\bigstar^{H_V}$ we can assume $W_1 \cap W_2 = \varnothing$. Indeed the constraint $\exists_W c$ is equivalent to $\exists_{W'} c[W \mapsto W']$, with W' made of fresh variables. We can hence assume that existential Herbrand constraints are renamed apart w.r.t. existential variables.

$$expand_x^{H_V}(\exists_W c) = \exists_W c,$$

$$rename_{x\to n}^{H_V}(\exists_W c) = \exists_W (c[x \mapsto n]).$$

The operations of Definition 3 are pointwise extended to $\wp(H_V)$. For instance, if $S_1, S_2 \subseteq H_V$, then $S_1 \bigstar^{H_V} S_2 = \{h_1 \bigstar^{H_V} h_2 \,|\, h_1 \in S_1,\ h_2 \in S_2 \text{ and } h_1 \bigstar^{H_V} h_2 \text{ is defined}\}$ and $\exists_x^{\wp(H_V)} S = \{\exists_x^{H_V} h \,|\, h \in S\}$. On the collecting domain $\wp(H_V)$ a new operation $\cup^{H_V}$ is defined as $\cup^{H_V}(S_1, S_2) = S_1 \cup S_2$.

We abuse notation and we use the operations of Definition 3 with sets of variables instead of single variables. For instance, $restrict_{\{x_1,\ldots,x_n\}}^{H_V}$ stands for the composition $restrict_{x_1}^{H_V} \cdots restrict_{x_n}^{H_V}$ and $expand_{x_1,\ldots,x_m\to n_1,\ldots,n_m}^{H_V}$ for the composition $expand_{x_1\to n_1}^{H_V} \cdots expand_{x_m\to n_m}^{H_V}$.

The *s-semantics* of a program is the least fixpoint of its *immediate consequence operator*.

Definition 4. Let P be a program over H. Its immediate consequence operator $T_P^H : \mathbb{I}^H \mapsto \mathbb{I}^H$ is such that

$$T_P^H(I)(\mathtt{p}) = \cup\left\{ restrict_{V\setminus\{X_1,\ldots,X_n\}}^{H_V}[[G]]I \,\middle|\, \mathtt{p}(X_1,\ldots,X_n) :- G. \in P \right\}$$

for every $\mathtt{p} \in \Pi$, where the *denotation* $[[G]]I$ of G in I is

$$[[G_1,\ldots,G_m]]I = [[G_1]]I \bigstar^{H_V} \cdots \bigstar^{H_V} [[G_m]]I$$

$$[[h]]I = \{h\} \quad \text{if } h \in H_V,$$

$$[[\mathtt{q}(Y_1,\ldots,Y_l)]]I = expand_{V\setminus\{Y_1,\ldots,Y_l\}}^{H_V} rename_{Z_1,\ldots,Z_l\to Y_1,\ldots,Y_l}^{H_{\{Z_1,\ldots,Z_l\}}} I(\mathtt{q})$$

if $\mathtt{q}(Z_1,\ldots,Z_l) : -G. \in P$.

As one can see from Definition 4, the denotation of the tail of a clause is computed by using the conjunction operator $\bigstar^{H_V}$ applied to the denotations of the components of the tail. The operator T_P then projects ($restrict^{H_V}$) this denotation over the variables that occur in the head of the clause. The denotation of a predicate $\mathtt{q}$ in the tail of a clause is computed by fetching its current interpretation, by renaming ($rename^{H_V}$) its variables in order to reflect its calling context and by enlarging ($expand^{H_V}$) the set of variables in order to cover the entire set V.

3.3. Abstract interpretation

Abstract interpretation [16,17] allows us to reason about the abstraction relation between two different domains (the *concrete* and the *abstract* domain).

We recall that a complete lattice L is a partially ordered set where least upper bound (or *join*, denoted by $\sqcup$) and greatest lower bound (or *meet*, denoted by $\sqcap$) exist for every subset of L. A *Moore family* M of C is a topped completely meet-closed subset of C, i.e., M contains the top element of C and is closed w.r.t. arbitrary meets. The *Moore* ($\sqcap_C$) *closure* of a set $A \subseteq C$ is denoted by $\mathrm{c}(A)$.

Definition 5. Let $\langle C, \leqslant \rangle$ and $\langle A, \preceq \rangle$ be two complete lattices (the concrete and the abstract domain). A *Galois connection* from C to A is a pair of monotonic maps $\alpha : C \to A$ (*abstraction*) and $\gamma : A \to C$ (*concretisation*) such that for each $x \in C$ we have $x \leqslant \gamma\alpha(x)$ and for each $y \in A$ we have $\alpha\gamma(y) \preceq y$. A *Galois insertion* is a Galois connection where $\alpha\gamma$ is the identity map on A.

The composition of Galois connections is a Galois connection. The composition of Galois insertions is a Galois insertion. A Galois connection is a Galois insertion if and only if γ is one-to-one or, equivalently, if and only if α is onto. In a Galois insertion, the abstraction map uniquely identifies the concretisation map and vice versa. It is well known [16] that the set of Galois insertions from C to A is isomorphic to the set of the Moore families of C. This means that every Moore family $M \subseteq C$ is an abstract domain whose concretisation map is the identity map. This way of looking at abstract domains allows us to distinguish the property of a domain from the properties of its representations.

Let $f : C^n \to C$ be a concrete operator and let $\tilde{f} : A^n \to A$. Then $\tilde{f}$ is a *correct approximation* of f if for all $y_1, \ldots, y_n \in A$ we have $\alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \preceq \tilde{f}(y_1, \ldots, y_n)$. For each operator f, there exists a *best correct* or *optimal approximation* $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$. The composition of correct approximations is a correct approximation but the composition of optimal approximations is not necessarily an optimal approximation. When f is clear from the context, we just say that $\tilde{f}$ is correct (optimal). The abstract domain A is called *condensing* w.r.t. $\tilde{f}$ if for every $x, y \in A$ we have $x \sqcap \tilde{f}(y) = \tilde{f}(x \sqcap y)$ [31,34].

Every abstract domain A, with abstraction function α^A, allows us to compute the corresponding abstract s-semantics of a logic program, by substituting A instead of H in Definitions 1–4. The denotation of a Herbrand constraint becomes its abstraction. Hence, we modify Definition 4 with $[[h]]I = \alpha^A(h)$. The precision of the abstract semantics (analysis) depends on the precision of the abstract domain.

3.4. Abstract compilation

As we said at the end of the previous section, we can compute the abstract s-semantics of a program by using its same definition instantiated over the abstract domain A. However, this requires to abstract the concrete constraints in the program at every iteration of the immediate consequence operator (Definition 4). It is hence natural to optimise the fixpoint computation by abstracting the logic program once and for all into an abstract logic program, and by then computing its s-semantics without using the abstraction function anymore. In such a case, Definition 4 can be instantiated to the abstract domain A without the modification described at the end of the previous section. This technique is called *abstract compilation* [12,26].

Example 6. The computation over A of the abstract s-semantics of the following logic program:

```
positive(X):-x=s(Y),integer(Y).   integer(X):-X=0.
                                  integer(X):-X=s(Y),integer(Y).
```

proceeds as follows. We first substitute the concrete constraints with their abstraction. Let $a_1 = \alpha^A(\mathtt{X} = \mathtt{s(Y)})$ and $a_2 = \alpha^A(\mathtt{X} = \mathtt{0})$. The compiled program is

```
positive(X): −a1, integer(Y).     integer(X): −a2.
                                  integer(X): −a1, integer(Y).
```

We then compute the fixpoint of the T_P^A operator (Definition 4).

Note that abstract compilation can be used only if all the abstract operations are defined over elements of the abstract domain only, which is what we assume when we instantiate Definition 3 over the abstract domain A. Instead, the conjunction operation of the domain *Sharing* of [31] is defined between a concrete element and an abstract one. This does not allow us to use abstract compilation for that domain. Actually, even Definition 4 must be modified to fit that domain.

3.5. Goal-independence

By *goal-independence* we mean that the (abstract) semantics of a program is computed for the most general goals only. The semantics of the other goals is derived from that of the most general goals by instantiation and without using the text of the program. Hence the semantics of the most general goals must contain all the information needed to derive the semantics of the other goals.

The advantage of goal-independence is that the analysis becomes naturally modular, since it cannot look at the text of other modules, but only at the summary information gained from them. Another advantage of goal-independence is that, once a module has been analysed, its source code can be kept secret. Thus the analysis can be applied also when the code cannot be publicly divulged for copyright reasons.

A typical example of a goal-independent analysis is that obtained through the computation of an abstract s-semantics (Section 3.2). Since only the most general goals are analysed, the analysis of a goal like `p(f(X))` is derived from the analysis of the most general goal `p(Y)` through its conjunction with the abstraction of $Y = \mathtt{f}(X)$.

In has been shown that, in general, a goal-independent analysis is less precise than a goal-dependent analysis computed by using the same abstract domain, and that, for domains which are condensing w.r.t. conjunction, both analyses have the same precision [25].

Note that our notion of *goal-independence* is different from that used in [9,18], where the program is still needed to derive the goal-dependent information from the (so-called) goal-independent analysis of the same program. Hence, our notion is in our opinion more correct.

3.6. Linear refinement

Given an abstract domain $A \subseteq C$, a domain refinement operator R yields an abstract domain $R(A) \subseteq C$ which is more precise than A, i.e., which contains A [19,23]. A classical domain refinement operator is the *reduced product* $A \sqcap B$ of two domains A and B, both contained in another domain C [16]. It is isomorphic to the Cartesian product of A and B, modulo the equivalence relation $\langle a_1, b_1\rangle \equiv \langle a_2, b_2\rangle$ if and only if $a_1 \sqcap b_1 = a_2 \sqcap b_2$. Hence pairs with the same *meaning* are identified.

Linear refinement [24] is a slight generalisation of Cousot's reduced power operation [16]. It allows us to include in a domain the information related to the propagation of the abstract property of interest before and after the application of a partial operator over C. We consider here just the case when $C = \wp(H_V)$ and the operator is the pointwise extension of conjunction (Definition 3).

Let $a, b \in \wp(H_V)$. We define the *linear refinement* of a w.r.t. b as

$$a \to b = \{h \in H_V \mid \text{if } a\bigstar^{H_V} h \text{ is defined then } a\bigstar^{H_V} h \leqslant b\}. \tag{2}$$

The set $a \to b$ contains exactly those existential Herbrand constraints which, upon conjunction with a constraint in a, become a constraint in b. If a and b are sets of constraints satisfying some property, you can view $a \to b$ as the set of constraints which transform the property a into the property b upon conjunction. An arrow $a \to b$ is called *tautological* when it coincides with H_V.

Example 7. For every $v \in V$, let $v = \{\exists_W c \in H_V \mid vars(c(v)) = \varnothing\}$. The set v is the set of constraints which bind v to a ground term. Let $x, y \in V$. Eq. (2) becomes in this case

$$\mathbf{x} \to \mathbf{y} = \left\{ \exists_W c \in H_V \;\middle|\; \begin{array}{l} \text{for all } \exists_{W'} c' \leqslant \exists_W c \\ \text{if } vars(c'(x)) = \varnothing, \text{then } vars(c(y)) = \varnothing \end{array} \right\}.$$

This means that every $h \in \mathbf{x} \to \mathbf{y}$ is such that in all its instantiations if x is ground then y is ground. Equivalently, you can say that h transforms the groundness of x into the groundness of y upon conjunction. For instance, we have $\{x \mapsto \mathtt{f}(y)\} \in \mathbf{x} \to \mathbf{y}$ and $\{y \mapsto \mathtt{f}(\mathtt{a})\} \in \mathbf{x} \to \mathbf{y}$. But $\{x \mapsto \mathtt{f}(\mathtt{a})\} \notin \mathbf{x} \to \mathbf{y}$. Note that, since groundness cannot be lost, $\mathbf{x} \to \mathbf{x} = H_V$. Hence $\mathbf{x} \to \mathbf{x}$ is a tautological arrow.

Given an abstract domain $L \subseteq H_V$, we define

$$L \triangleright L = \mathrm{c}\{a \to b \mid a, b \in L\}. \tag{3}$$

The set $L \triangleright L$ is then the collection of all possible intersections of arrows which can be built from elements of L. Note that $l \to (a \sqcap b) = (l \to a) \sqcap (l \to b)$.

The *linear refinement* $L \to L$ of L is the domain

$$L \to L = L \sqcap (L \triangleright L). \tag{4}$$

If $L \subseteq L \triangleright L$, i.e., if the properties in L are (degenerate) cases of intersections of arrows, (4) can be simplified into

$$L \to L = L \triangleright L. \tag{5}$$

This simplification is relevant since it allows a simpler representation and simpler operations for $L \to L$. Indeed, we need to represent elements and operations over $L \triangleright L$ instead of elements and operations over the reduced product $L \sqcap (L \triangleright L)$.

Example 8. By using the notation of Example 7, the set $\{v \mid v \in V\}$ is able to express basic facts about the groundness of single variables. The domain $G = \mathrm{c}\{v \mid v \in V\}$ is able to express basic facts about the groundness of sets of variables. For instance, $\mathbf{x} \cap \mathbf{y}$ (which, in groundness analysis, is traditionally written as $\mathbf{xy}$ [2,14,15,37]) is the set of constraints where both x and y are ground, for every $x, y \in V$. It has been proved [37] that the traditional domains *Def* and *Pos* for groundness analysis [2,14,15] can be derived from G through linear refinement. Namely, we have $Def = G \to G$ and $Pos = Def \to Def$. Moreover, *Pos* cannot be further linearly refined, i.e., $Pos \to Pos = Pos$. We have $G \subseteq G \triangleright G$. In particular, given $\{x_1, \ldots, x_n\} \subseteq V$, it can be shown that $\mathbf{x}_1 \cdots \mathbf{x}_n = H_V \to \mathbf{x}_1 \cdots \mathbf{x}_n$, where $H_V = \cap\{\} \in G$. That is, the variables $x_1, \ldots, x_n$ are ground in $h \in H_V$ if and only if they are ground in every instance of h.

4. Pair-independence analysis

The traditional domain for sharing or set-dependence analysis [28,31] is $Sharing = \wp(\wp(V))$. For instance, the Herbrand constraint $\{x = \mathtt{f}(y), z = \mathtt{g}(y,v)\}$ is abstracted into $\{\varnothing, \{x,y,z\}, \{v,z\}\}$, which represents the fact that x, y, and z share y, while v and z share v. The empty set is a consequence of the fact that no variable shares x.

It has been noted that the applications of dependence analysis always consider dependence information about pairs of variables, and that information is redundant for pair-dependence information [4]. Therefore, in the following we consider pair-dependence information, i.e., we compute a superset of the set of pairs of variables which actually share in a given program point. This is exactly the same as computing a subset of the set of pairs of variables which are guaranteed not to share in a given program point. We prefer this second point of view, since definite information is more intuitive than possible information. The simplest definition of an abstract domain for pair-independence coincides with the same property we want to observe.

We define now the set $(\mathbf{v_1}, \mathbf{v_2})_V$ of the existential Herbrand constraints in which the variables v_1 and v_2 are independent, and the abstract domain $Indep_V$ which expresses properties of independence.

Definition 9. Let $\{v_1, v_2\} \subseteq V$. We define

$$(\mathbf{v_1}, \mathbf{v_2})_V = \{\exists_W c \in H_V \mid vars(c(v_1)) \cap vars(c(v_2)) = \varnothing\},$$

$$Indep_V = \mathrm{c}\{(\mathbf{v_1}, \mathbf{v_2})_V \mid \{v_1, v_2\} \subseteq V\}$$

ordered by set inclusion. When the set V is obvious from the context, we write $(\mathbf{v_1}, \mathbf{v_2})$ for $(\mathbf{v_1}, \mathbf{v_2})_V$. Let $\{(v_1, v_2'), \ldots, (v_n, v_n')\} \subseteq V \times V$ and $P \subseteq V \times V$. We write $(\mathbf{v_1}, \mathbf{v_1'}) \cdots (\mathbf{v_n}, \mathbf{v_n'})$ for $\cap\{(\mathbf{v_i}, \mathbf{v_i'}) \mid i = 1, \ldots, n\}$ and $\mathbf{P}$ for $\cap\{\mathbf{p} \mid p \in P\}$.

Example 10. Let $V = \{v, x, y, z\}$ and $h = \{x = \mathtt{f}(y), z = \mathtt{g}(y,v)\}$. We have $h \in (\mathbf{v}, \mathbf{x})$ since x and v do not share any variable in h. But $h \notin (\mathbf{y}, \mathbf{z})$, since z and y share y in h. The abstraction[2] of h is $(\mathbf{v}, \mathbf{x})(\mathbf{v}, \mathbf{y})$.

Note that, if $v \in V$, then $(\mathbf{v}, \mathbf{v}) = \{\exists_W c \in H_V \mid vars(c(v)) = \varnothing\}$, i.e., $(\mathbf{v}, \mathbf{v})$ is the set of existential Herbrand constraints where v is ground.

The domain $Indep_V$ is isomorphic to the domain for pair-sharing defined in [4] as an abstraction of the larger domain for set-sharing analysis presented in [28,31].

The optimal approximation of concrete conjunction is indeed very poor w.r.t. precision. This is because $Indep_V$ is not precise enough to distinguish concrete substitutions whose behaviour, w.r.t. concrete conjunction, is quite different.

Example 11. Assume that $\{x, y, z\} \subseteq V$. The constraints $\varnothing$ and $h = \exists_{\{w\}}\{x = \mathtt{f}(w, w)\}$ are both abstracted into the element $\cap\{(\mathbf{v_1}, \mathbf{v_2}) \mid \{v_1, v_2\} \subseteq V \text{ and } v_1 \neq v_2\}$ of $Indep_V$. However, they have very different sharing behaviour when conjuncted with $\{x = \mathtt{f}(y, z)\}$. Indeed, in the first case y and z do not share after the conjunction, while in the second case they do.

[2] The abstraction map is induced by the abstract domain $Indep_V$ [16].

This imprecision was avoided in [4,28,31] through the use of a hybrid conjunction procedure, which computes the conjunction of an abstract element and a *concrete* element. Hence it allows us to distinguish $\varnothing$ from h, because the hybrid conjunction works explicitly with them, and not with their abstractions. However, this approach does not allow us to apply abstract compilation. Moreover, the approach of [4,28], even by using the hybrid unification procedure, is sometimes imprecise.

Example 12. Assume that $\{v,x,y,z\} \subseteq V$. The sharing domains of [4,28,31] cannot distinguish between $h_1 = \{x = \mathtt{f}(y), v = \mathtt{f}(z)\}$ and $h_2 = \{x = \mathtt{f}(y), v = \mathtt{g}(z)\}$. However, they have very different sharing behaviour when conjuncted with $\{x = v\}$. Namely, the fact that y and z are made to share depends on whether h_1 or h_2 is conjuncted with $\{x = v\}$.

In an attempt to solve these problems, we can linearly refine $Indep_V$ w.r.t. concrete conjunction, as it has been done in the case of groundness [24,37]. Consider the refined domain $Indep_V^1 = Indep_V \to Indep_V$. The domain $Indep_V^1$ is precise enough to solve the problem in Example 11.

Example 13. Consider Example 11. The constraint $\varnothing$ belongs to $(\mathbf{y},\mathbf{z}) \to (\mathbf{y},\mathbf{z})$, while h does not. Indeed, the conjunction of every $h' \in H_V$ where y and z do not share with the Herbrand constraint $\varnothing$ cannot make y and z to share. Instead, the independence of y and z in $h'' = \{x = \mathtt{f}(y,z)\}$ is lost when we conjunct h'' with h.

However, $Indep_V^1$ is not precise enough yet. For instance, it cannot solve the problem shown by Example 12. This negative result is not very relevant, since the imprecision shown by Example 12 does not affect the usefulness of the analysis (the analysis described in [31,36] has been used with good results, though it cannot solve this problem). Instead, the following example shows another source of imprecision.

Example 14. Let $V = \{v,x,y,z\}$, $h_1 = \{x = \mathtt{f}(y,z)\}$, and $h_2 = \{x = v\}$. The optimal approximation $\bigstar^{Indep_V^1}$ of concrete conjunction is such that $\alpha(h_1)\bigstar^{Indep_V^1}\alpha(h_2) \not\subseteq (\mathbf{y},\mathbf{z})$ (see the proof in Appendix A), i.e., it is not able to conclude that y and z do not share after concrete conjunction of h_1 and h_2 (this leads to a very imprecise analysis).

Since $Indep_V \subseteq Indep_V^1$ and since, as proved in [4], $Indep_V$ is as precise as Jacobs and Langen's domain [28] w.r.t. pair-(in)dependence information, we conclude that $Indep_V^1$ is strictly more precise than $Indep_V$ and *Sharing* (Example 13), but it is sometimes too imprecise, and the same happens in such a case for $Indep_V$ and *Sharing* (Example 14). Note that, at the theoretical level, the precision of the analysis depends on the abstract domain only. Everything else (abstraction map and abstract operations) is induced by the abstract domain.

Another problem with the domain $Indep_V$ is that we cannot use the simplified equation (5), as Proposition 15 shows. This complicates its implementation.

Proposition 15. *If* $\#V \geqslant 3$, *then* $Indep_V \not\subseteq Indep_V \rhd Indep_V$.

5. Freeness analysis

The simplest domain for freeness analysis coincides with the same property. We hence define the set v_V of the existential Herbrand constraints where v is free and the abstract domain $Indep_V$ which expresses properties of freeness.

Definition 16. Let $v \in V$. We define

$$\mathbf{v}_V = \{\exists_W c \in H_V \mid c(v) \in V \cup W\},$$
$$Free_V = \mathrm{c}\{\mathbf{v}_V \mid v \in V\}$$

ordered by set inclusion. When V is obvious from the context, we write $\mathbf{v}$ for $\mathbf{v}_V$. Let $\{v_1, \ldots, v_n\} \subseteq V$ and $F \subseteq V$. We write $\mathbf{v}_1 \cdots \mathbf{v}_n$ for $\cap\{\mathbf{v_i} \mid i = 1, \ldots, n\}$ and $\mathbf{F}$ for $\cap\{\mathbf{v} \mid v \in F\}$.

Example 17. Let $V = \{k, x, y, z\}$ and $h = \exists_{\{w\}}\{x = \mathtt{f}(y), z = w\}$. We have $h \in \mathbf{kyz}$, since k, y, and z are free in h, but $h \notin \mathbf{kxyz}$, since x is not free in h.

The domain $Free_V$ is not precise enough for a practical use.

Example 18. Let $\bigstar^{Free_V}$ be the optimal approximation on $Free_V$ of concrete conjunction. Consider the two existential Herbrand constraints $h_1 = \{x = \mathtt{a}\}$ and $h_2 = \{y = \mathtt{a}\}$ over $V = \{x, y, z\}$. Their concrete conjunction leaves z free. A useful domain for freeness analysis must capture this behaviour. However, $\alpha(h_1)\bigstar^{Free_V}\alpha(h_2) \not\subseteq \mathbf{z}$. Indeed, in $Free_V$, h_1 is abstracted to $\mathbf{yz}$ and h_2 to $\mathbf{xz}$. We have $\mathbf{yz}\bigstar^{Free_V}\mathbf{xz} = H_V$ which is not contained in $\mathbf{z}$. This is because $\{x = \mathtt{a}, y = z\}$ is abstracted to $\mathbf{yz}$ as h_1, and its conjunction with h_2 leaves z non-free.

To solve this problem, we can try to linearly refine the domain $Free_V$. However, the following result shows that the refined domain $Free_V^1 = Free_V \to Free_V$ does not solve the problem shown in Example 18.

Example 19. Let $\bigstar^{Free_V^1}$ be the optimal approximation on $Free_V^1$ of concrete conjunction. Let h_1 and h_2 be as in Example 18. We have $\alpha(h_1)\bigstar^{Free_V^1}\alpha(h_2) \not\subseteq \mathbf{z}$ (see the proof in Appendix A).

Moreover, we cannot use the simplified Equation (5). This complicates its implementation.

Proposition 20. *If* $\#V \geqslant 2$, *then* $Free_V \not\subseteq Free_V \rhd Free_V$.

In [27] it is shown how to overcome these problems by using only freeness information. However, the solution is not general enough and proofs are very complex.

6. Linear refinement revisited

Example 14 shows that the refinement of the basic domain for pair-independence analysis is not precise enough. Example 19 shows that the refinement of the basic domain for freeness analysis is

not precise enough too. If we are interested in improving the precision of pair-independence or freeness analysis we have two possibilities. The first is to perform further refinements, hopefully forcing the imprecision to disappear. The second is to redesign the basic domain, adding the information which is needed in order to obtain useful arrows by refinement.

This last alternative does not look attractive at a first glance. Linear refinement was in fact originally presented as an *automatic* methodology, able to improve the precision of the abstract version of a given concrete operation [24]. Instead, the second approach reintroduces a non-methodological choice about the information which is needed in order to obtain more useful arrows. However, we want to convince the reader that in this case the first approach is the wrong one, at least if we are interested in an abstract domain with a *computationally interesting* representation and simple *algorithmic* definitions for its abstract operators.

It is true indeed that the refinement of an already refined domain admits a very simple representation as arrows of arrows. However, this representation is huge and impractical, and we have no guarantee that it will eventually reach a useful precision. Moreover, as we perform more refinements, the abstraction function and the abstract operators are more difficult to devise, and computationally more expensive. Note that these remarks are not always true (consider groundness analysis [37], for instance).

In Example 14, we would like to approximate h_1 with the arrow $\mathbf{x}(\mathbf{y},\mathbf{z}) \rightarrow (\mathbf{y},\mathbf{z})$, since if x is free and y and z are independent in h_2 then y and z are independent in $h_1 \bigstar^{H_V} h_2$. We think that freeness information cannot be expressed as a property of independence of variables, however complex it might be. In any case, this question has little practical interest since, by following the second approach outlined above, we can allow freeness information to appear in the left of arrows for independence information.

Similarly, in Example 18 we would like to approximate h_1 with the arrow $\mathbf{z}(\mathbf{x},\mathbf{z}) \rightarrow \mathbf{z}$, since if z is free and x and z are independent in h_2 then z is free in $h_1 \bigstar^{H_V} h_2$. By following the second approach outlined above, we can allow independence information to appear in the left of arrows for freeness information.

Formally, the above remarks mean that we want to compute the linear refinement of the reduced product of $Indep_V$ with $Free_V$. We have hence provided inside the domain refinement framework a reformulation of the already known result about sharing/freeness interaction [31,36]. We think that the domain refinement theory gives to this problem a better perspective and a greater generality.

7. Pair-independence and freeness

In the previous section, we have shown that it is very natural to define a domain for pair-independence and freeness analysis in such a way that its linear refinement is precise enough to solve the problems of Examples 14 and 18. The definition below formalises this idea.

Definition 21. We define $IndepFree_V = Indep_V \sqcap Free_V$ and $IndepFree_V^1 = IndepFree_V \rightarrow IndepFree_V$.

We extend the notation already introduced for the elements of $Indep_V$ and $Free_V$ (Definitions 9 and 16). For instance, $\mathbf{x}(\mathbf{y},\mathbf{z})$ stands for $\mathbf{x} \cap (\mathbf{y},\mathbf{z})$.

The following result shows that we can use Eq. (5), which will simplify the implementation of $IndepFree_V^1$.

Proposition 22. *We have* $IndepFree_V \subseteq IndepFree_V \triangleright IndepFree_V$.

Proof. We prove that every $e \in IndepFree_V$ can be written as $X \to e$ for a suitable $X \in IndepFree_V$. Let $X = \mathbf{V} \cap A$, where $A = \cap\{(\mathbf{v_1}, \mathbf{v_2}) \mid \{v_1, v_2\} \subseteq V, v_1 \neq v_2\}$. The set X collects all existential Herbrand constraints where all variables are free and mutually independent. It can be easily seen that (up to equivalence) $X = \{\varnothing\}$. By definition, we have $X \in IndepFree_V$. Moreover, since $\varnothing$ cannot change the freeness or independence properties of a constraint, we have $X \to e = e$ for every $e \in IndepFree_V$. □

Therefore, $IndepFree_V \subseteq IndepFree_V^1 = IndepFree_V \triangleright IndepFree_V$. In general, this inclusion is strict, as shown below.

Proposition 23. *The domain* $IndepFree_V^1$ *contains information about functors and linearity not contained in* $IndepFree_V$.

Proof. *Functor names.* Let h_1 and h_2 be as in Example 12. They cannot be distinguished in $IndepFree_V$, but h_2 belongs to $\mathbf{xv}(\mathbf{x}, \mathbf{z})(\mathbf{y}, \mathbf{v})(\mathbf{y}, \mathbf{z}) \to (\mathbf{y}, \mathbf{z})$, while h_1 does not.[3]

Linearity. Let $\{v, x, y\} \subseteq V$, $h_1 = \exists_{\{w_1, w_2\}}\{v = \mathtt{f}(w_1, w_2)\}$ and $h_2 = \exists_{\{w\}}\{v = \mathtt{f}(w, w)\}$. The constraint h_1 binds v to a linear term, while h_2 binds v to a non-linear term. The constraints h_1 and h_2 cannot be distinguished in $IndepFree_V$. In $IndepFree_V^1$, instead, $h_1 \in (\mathbf{x}, \mathbf{y}) \to (\mathbf{x}, \mathbf{y})$ while $h_2 \notin (\mathbf{x}, \mathbf{y}) \to (\mathbf{x}, \mathbf{y})$. □

The additional information contained in $IndepFree_V^1$ is relevant for pair-independence analysis. Namely, functor names are important since we do not need to know that x and y do not share in h in order to conclude that v and v' do not share in $h \bigstar^{H_V} \{x = \mathtt{f}(v), y = \mathtt{g}(v')\}$, provided that x and y are both free in h. Linearity information is important since even if we know that x is free and y and z do not share in some constraint h, we cannot conclude that y and z do not share in $h \bigstar^{H_V} h_1$, where $h_1 = \exists_{\{w\}}\{x = \mathtt{f}(y, z), v = \mathtt{g}(w, w)\}$ (consider $h = \{v = \mathtt{g}(y, z)\}$), while we can, if we take $h_2 = \exists_{\{w_1, w_2\}}\{x = \mathtt{f}(y, z), v = \mathtt{g}(w_1, w_2)\}$ in $h \bigstar^{H_V} h_2$. In $IndepFree_V^1$, this is captured by the fact that $h_1 \notin \mathbf{x}(\mathbf{y}, \mathbf{z}) \to (\mathbf{y}, \mathbf{z})$, while $h_2 \in \mathbf{x}(\mathbf{y}, \mathbf{z}) \to (\mathbf{y}, \mathbf{z})$.

8. A representation

We define here a data structure to represent the elements of $IndepFree_V^1$. Moreover, we define algorithms over this data structure which can be used to approximate the abstract operations induced over $IndepFree_V^1$ by the concrete operations of Definition 3. We will show why we claim that this representation can be more generally applied to every linearly refined domain.

[3] It does actually belong to $(\mathbf{x}, \mathbf{y})(\mathbf{x}, \mathbf{z})(\mathbf{y}, \mathbf{v})(\mathbf{y}, \mathbf{z}) \to (\mathbf{y}, \mathbf{z})$.

8.1. The data structure

The obvious representation for the elements of $IndepFree^1_V$ is made of arrows between sets of free variables and of unordered pairs of independent variables.

Definition 24. Let $V_2 = \{P \subseteq V \mid \#P = 2\}$ and $S = \{p_1 \cdots p_n \mid p_i \in V \cup V_2 \text{ for } i = 1, \ldots, n\}$. We write the elements of V_2 as (v_1, v_2), with $v_1, v_2 \in V$. The set S expresses which variables are free and which pairs of variables are independent. Let $Rep_V = \wp(\{l \Rightarrow r \mid l, r \in S\})$. We define the concretisation map $\gamma_{Rep} : Rep_V \mapsto IndepFree^1_V$ as

$$\gamma_{Rep}(A) = \cap\{\mathbf{l} \to \mathbf{r} \mid l \Rightarrow r \in A\}.$$

Example 25. Let $v, x, y \in V$. The element $x \in S$ means that x is free. The element $(x, v) \in S$ means that x and v are independent. The element $x(x, v) \in S$ means that x is free and x and v are independent. We have $x(x, v)(x, y) \Rightarrow (x, y) \in Rep_V$.

More generally, if an abstract domain D contains elements $p_1, \ldots, p_n$, we can represent elements of the linear refinement domain $D^1 = D \to D$ by arrows between subsets of $p_1, \ldots, p_n$.

Definition 26. Let $s \in S$ and $A \in Rep_V$. Their *dimension* is defined as $dim(s) = \#s$ and $dim(A) = \sum_{l \Rightarrow r \in A}(dim(l) + dim(r))$.

Let $A \in Rep_V$. The set of variables which are free and that of the pairs of variables which are independent in every existential Herbrand constraint which belongs to the concretisation of A can be under approximated through the maps $free_V$ and $indep_V$.

Definition 27. We define the *extractors* $free_V : Rep_V \mapsto \wp(V)$ and $indep_V : Rep_V \mapsto V_2$ as

$$free_V(A) = \{v \in V \mid l \Rightarrow v \in A \text{ and } v'(v', v') \not\subseteq l \text{ for any } v' \in V\},$$
$$indep_V(A) = \{(v_1, v_2) \in V_2 \mid l \Rightarrow (v_1, v_2) \in A \text{ and } (v, v) \notin l \text{ for any } v \in V\}.$$

Example 28. Let $V = \{v, x, y, z\}$ and

$$A = \{v(v, x) \Rightarrow v, v(v, v) \Rightarrow x, x(x, z) \Rightarrow (x, z), (v, v) \Rightarrow (y, z)\}.$$

We have $free_V(A) = \{v\}$ and $indep_V(A) = \{(x, z)\}$.

Extractors like those in Definition 27 can be more generally defined by considering which arrows in A have their tail satisfied by the empty existential Herbrand constraint. Their heads must hold for the constraints approximated by A. Knowledge about the specific abstract domain can then improve this definition.

The following result shows that the variables in $free_V(A)$ are actually free and the pairs in $indep_V(A)$ are actually independent in the concretisation of A.

Proposition 29. *Let $A \in Rep_V$. Then $\gamma_{Rep}(A) \subseteq \mathbf{free_V}(\mathbf{A})$ and $\gamma_{Rep}(A) \subseteq \mathbf{indep_V}(\mathbf{A})$.*

8.2. The abstract operators

Consider the conjunction operation. Since Rep_V is made of arrows, i.e., dependences, abstract conjunction becomes folding of arrows.

Definition 30. Let $T \in Rep_V$ *tautological*, i.e., such that $\gamma_{Rep}(A)$ is tautological[4] (Section 3.6). Let $A_1, A_2 \in Rep_V$. We define

$$A_1 \bigstar^{Rep_V} A_2 = \left\{ l_1 \cdots l_n \Rightarrow r \;\middle|\; \begin{array}{l} r_1 \cdots r_n \Rightarrow r \in A_2 \cup T,\ l_i \Rightarrow r'_i \in A_1 \cup T \\ \text{and } \mathbf{r}'_i \subseteq \mathbf{r}_i \text{ for } i = 1, \ldots, n \\ \\ \text{or} \\ \\ r_1 \cdots r_n \Rightarrow r \in A_1 \cup T,\ l_i \Rightarrow r'_i \in A_2 \cup T \\ \text{and } \mathbf{r}'_i \subseteq \mathbf{r}_i \text{ for } i = 1, \ldots, n. \end{array} \right\}.$$

Definition 30 holds for every linearly refined domain, since it is based only on the notion of arrow. However, specific sets of tautological arrows must be used.

The notion of *entailment* ($\subseteq$) in Definition 30 is semantical. Our implementation (Section 10) uses instead syntactical identity, which is of course correct but incomplete w.r.t. semantical entailment. It moreover uses $T = \{(v,v) \Rightarrow (v,v) \,|\, v \in V\}$. Those tautological arrows mean that the groundness of a variable cannot be lost. The following example shows that by using this set instead of $T = \varnothing$ we obtain a better precision for the abstract conjunction.

Example 31. Let $T = \varnothing$ in Definition 30. Let $V = \{x, y, z\}$, $A_1 = \{xy \Rightarrow x, (x,y)(x,z) \Rightarrow (x,y), (x,x) \Rightarrow (z,z)\}$, and $A_2 = \{x(x,y) \Rightarrow x, \Rightarrow (y,y)\}$. Then

$$A_1 \bigstar^{Rep_V} A_2 = \{xy(x,y)(x,z) \Rightarrow x, \Rightarrow (y,y)\}.$$

We do not obtain the arrow $\{(x,x) \Rightarrow (z,z)\}$, although it must hold for $A_1 \bigstar^{Rep_V} A_2$, since it holds for A_1 and groundness dependences cannot be lost. If we let $T \supseteq \{(v,v) \Rightarrow (v,v) \,|\, v \in V\}$, that arrow would be included in the conjunction.

Proposition 32. *The operation* $\bigstar^{Rep_V}$ *is correct and has worst-case time complexity* $O((a_1 + t)(a_2 + t)k)$*, where* a_1*,* a_2*, and* t *are the dimensions of* A_1*,* A_2*, and* T*, respectively, and* k *is the complexity of the* $\subseteq$ *test.*

The maps $restrict_x^{H_V}$, $expand_x^{H_V}$, and $rename_{x \to n}^{H_V}$ induce three maps on Rep_V.

Definition 33. Let $x \in V$, $n \notin V$, $X = \{x\} \cup \{(v,x) \,|\, v \in V \setminus x\}$, and $A \in Rep_V$. We define

$$restrict_x^{Rep_V}(A) = \left\{ l \setminus X \Rightarrow r \;\middle|\; \begin{array}{l} l \Rightarrow r \in A,\ (x,x) \notin l,\ r \not\equiv x, \\ r \not\equiv (x,v) \text{ for every } v \in V \end{array} \right\},$$

$$rename_{x \to n}^{Rep_V}(A) = A[x \mapsto n].$$

[4] The larger T is, the more precise $\bigstar^{Rep_V}$ is.

While the definition of renaming is independent of the abstract domain D, that of restriction depends on it. However, it can be derived almost automatically from D. Namely, in general we must remove those arrows whose head contains x, which is not used anymore. If instead x occurs in the tail of an arrow, we can remove that occurrence as long as the property it expresses is true in a constraint where x does not occur. This is the case of the properties in the set X of Definition 33. Otherwise, that property will never be satisfied, the tail of the arrow is always false and we can safely remove it.

Example 34. Let $V = \{x, y, z\}$, $n \notin V$, and $A = \{xy \Rightarrow x, xy \Rightarrow y, (x,x) \Rightarrow (y,z), (y,z)(x,y)(x,z) \Rightarrow (y,z)\}$. Then

$$restrict_x^{Rep_V}(A) = \{y \Rightarrow y, (y,z) \Rightarrow (y,z)\},$$

$$rename_{x\to n}^{Rep_V}(A) = \{ny \Rightarrow n, ny \Rightarrow y, (n,n) \Rightarrow (y,z), (y,z)(n,y)(n,z) \Rightarrow (y,z)\}.$$

Proposition 35. *The maps $restrict_x^{Rep_V}$ and $rename_{x\to n}^{Rep_V}$ are correct and have worst-case time complexity linear in the dimension of their operand.*

Let $x \notin V$. The $expand_x^{\wp(H_V)}$ operation maps a subset of H_V in the same subset of $H_{V\cup x}$. We now show that if an arrow is correct for $h \in H_V$, then it is correct for $expand_x^{H_V}(h)$.

Lemma 36. *Let $\{\mathbf{l}_V, \mathbf{r}_V\} \subseteq IndepFree_V$ and $x \in \mathcal{V} \setminus V$. Then $\mathbf{l}_V \to \mathbf{r}_V \subseteq \mathbf{l}_{V\cup x} \to \mathbf{r}_{V\cup x}$.*

Lemma 36 entails that the arrows of $A \in Rep_V$ can be put in $expand_x^{Rep_V}(A)$. We wonder however whether there are other arrows in $expand_x^{Rep_V}$, possibly involving the newly introduced variable x. The lemma below shows that the behaviour of two variables which do not occur in an existential Herbrand constraint is the same.

Lemma 37. *Let $\{v_1, v_2\} \subseteq V$, $\mathbf{l} \to \mathbf{r} \in IndepFree_V \to IndepFree_V$, and $\exists_W c \in H_V$ such that $\{v_1, v_2\} \cap (dom(c) \cup rng(c)) = \varnothing$. We have $\exists_W c \in \mathbf{l} \to \mathbf{r}$ if and only if $\exists_W c \in \mathbf{l}[\mathbf{v_2} \mapsto \mathbf{v_1}, \mathbf{v_1} \mapsto \mathbf{v_2}] \to \mathbf{r}[\mathbf{v_2} \mapsto \mathbf{v_1}, \mathbf{v_1} \mapsto \mathbf{v_2}]$.*

Lemma 37 suggests us to use a distinguished variable $? \in V$ which does not occur in the programs. This means that it does not occur in the existential Herbrand constraints we are dealing with. Whenever we need to know the behaviour of a newly introduced variable x, like in $expand_x^{Rep_V}(A)$, we can look at the arrows in A having ? on the right and *incarnate* them into new arrows, obtained by substituting x for ?. The use of this generic variable is efficient and does not affect the clean construction through linear refinement. However, since pair-independence is a property of pairs of variables, when we incarnate ? into x we wish to know how this new x behaves in conjunction with the distinguished variable itself. This means that we need to know the behaviour of a variable which does not occur in the program w.r.t. another variable which does not occur in the program. This is possible if we use two distinguished variables $?_1$ and $?_2$. This argument leads to the following definition.

Definition 38. Let $x \in \mathcal{V} \setminus V$, $\{?_1, ?_2\} \subseteq V$, and $A \in Rep_V$. We define

$$expand_x^{Rep_V}(A) = A \cup \{l[?_1 \mapsto x, ?_2 \mapsto ?_1] \Rightarrow r[?_1 \mapsto x, ?_2 \mapsto ?_1] \mid l \Rightarrow r \in A\}.$$

Definition 38 can be generalised to every abstract domain D. We just need to consider as many ?-variables as they are used by the properties expressed by D.

Example 39. Let $V = \{x, y, z\}$, $n \in \mathcal{V} \setminus V$, and

$$A = \{?_1(x, ?_1) \Rightarrow ?_1, (?_1, x)(?_1, z)(?_1, ?_2) \Rightarrow (?_1, ?_2), xyz \Rightarrow z\}.$$

We have

$$expand_n^{Rep_V}(A) = \left\{ \begin{array}{l} xyz \Rightarrow z, n(x, n) \Rightarrow n, (n, x)(n, z)(n, ?_1) \Rightarrow (n, ?_1), \\ ?_1(x, ?_1) \Rightarrow ?_1, (?_1, x)(?_1, z)(?_1, ?_2) \Rightarrow (?_1, ?_2) \end{array} \right\}.$$

Proposition 40. *Let $x \in \mathcal{V} \setminus V$. Then $expand_x^{Rep_V}$ is correct w.r.t. $expand_x^{\wp(H_V)}$ applied to sets of existential Herbrand constraints where $?_1$ and $?_2$ do not occur. Its worst-case time complexity is linear in the dimension of its argument.*

Proposition 40 is a weak correctness result for $expand_x^{Rep_V}$. It is sufficient for our purposes, since we assume that programs do not contain $?_1$ and $?_2$.

The map $\cup^{\wp(H_V)}$ induces a map on Rep_V which can be approximated by merging the tails of the arrows with the same head. This operation is independent of the abstract domain, since it is based only on the notion of arrow.

Definition 41. Let $A_1, A_2 \in Rep_V$. Then

$$\bigcup^{Rep_V}(A_1, A_2) = \{l_1 l_2 \Rightarrow r \mid l_1 \Rightarrow r \in A_1 \text{ and } l_2 \Rightarrow r \in A_2\}.$$

Proposition 42. *The map $\bigcup^{Rep_V}$ is correct and has worst-case time complexity $O(a_1 a_2)$, where a_1 and a_2 are the dimensions of its arguments.*

9. Abstraction

Let $h \in H_V$. In this section we provide an algorithm for computing an approximation of $\alpha_{Rep}\alpha_{IndepFree^1}(h)$. The problem consists in finding all the arrows $l \Rightarrow r$, with r singleton, such that $h \in \mathbf{l} \to \mathbf{r}$. Of course, we do not need to consider tautological arrows like $(v, v) \Rightarrow (v, w)$. For a domain as complex as $IndepFree_V^1$, the solution of this problem is too ambitious. This is why we provide just a correct approximation of $\alpha_{Rep}\alpha_{IndepFree^1}(h)$. To this purpose, we follow a methodological approach which can be useful also for other domains. Namely, a correct approximation of $\alpha_{Rep}\alpha_{IndepFree^1}(\exists_{Wc})$ is computed as the abstract conjunction of the abstraction of every single binding in c. The result can be later strengthened with arrows which can be derived by looking globally at c. Note that this approach leads to the optimal abstraction map in the case of groundness analysis, while it leads to a correct but non-necessarily optimal abstraction map in our case.

9.1. The abstraction of a single binding

Let $v = t$, with $v \in V$ and $t \in terms(\Sigma, V)$, be a single binding. Its abstraction consists in three sets of arrows, i.e., arrows for groundness, arrows for pair-independence, and arrows for freeness. Note that the arrows for groundness are a particular case of those for pair-independence, but it is simpler to deal with them separately.

Definition 43. Let $v \in V$ and $t \in terms(\Sigma, V)$. We define

$$\alpha^V_{alg}(v = t) = \alpha^V_{gr}(v = t) \cup \alpha^V_{indep}(v = t) \cup \alpha^V_{fr}(v = t),$$

where the functions α^V_{gr}, α^V_{indep}, and α^V_{fr} are defined in Figs. 1–3, respectively. In those figures we write $t(v_1, \ldots, v_n)$ for $t \in terms(\Sigma, V)$ whenever $vars(t) = \{v_1, \ldots, v_n\}$. When $n = 0$ then $t(v_1, \ldots, v_n)$ is a ground term. Moreover, we assume variables with different names to be different variables.

The arrows in $\alpha_{alg}(z = t)$ are correct for $\{z = t\} \in C_V$.

Lemma 44. *Let $z \in V$ and $t \in terms(\Sigma, V)$. Then $\{z = t\} \in \gamma_{Rep}(\alpha_{alg}(z = t))$.*

$$\alpha^V_{gr}(v = t(v_1, \ldots, v_n)) = \bigcup_{v' \in V} \left\{ \begin{array}{l} (v_1, v_1) \cdots (v_n, v_n) \Rightarrow (v, v'), \\ (v, v) \Rightarrow (v_1, v'), \\ \quad \vdots \\ (v, v) \Rightarrow (v_n, v') \end{array} \right\}$$

Fig. 1. The abstraction rules for groundness.

$$\begin{aligned}
\alpha^V_{indep}(x = t) &= \bigcup_{\{v,v'\} \subseteq V} \alpha^{(v,v')}_{indep}(x = t)) \\
\alpha^{(v,v')}_{indep}(v = t(v', v_1, \ldots, v_n)) &= \{\} \\
\alpha^{(v,v')}_{indep}(x = t(v, v', v_1, \ldots, v_n)) &= \{x(v, v') \Rightarrow (v, v')\} \\
\alpha^{(v,v')}_{indep}(v = t) &= \{\} \quad t \text{ ground} \\
\alpha^{(v,v')}_{indep}(v = t(v_1, \ldots, v_n)) &= \{(v', v)(v', v_1) \cdots (v', v_n) \Rightarrow (v', v)\} \quad t(v_1, \ldots, v_n) \text{ non-ground} \\
\alpha^{(v,v')}_{indep}(x = t(v, v_1, \ldots, v_n)) &= \left\{ \begin{array}{l} (v', v)(v', x)(v', v_1) \cdots (v', v_n) \Rightarrow (v', v), \\ (v', v)(v', x)x \Rightarrow (v', v) \end{array} \right\} \\
\alpha^{(v,v')}_{indep}(x = t) &= \{(v, v') \Rightarrow (v, v')\} \quad t \text{ ground} \\
\alpha^{(v,v')}_{indep}(x = t(v_1, \ldots, v_n)) &= \left\{ \begin{array}{l} (v, v')(v, x)(v, v_1) \cdots (v, v_n) \Rightarrow (v, v'), \\ (v, v')(v', x)(v', v_1) \cdots (v', v_n) \Rightarrow (v, v'), \\ (v, v')(v, x)(v', x)x \Rightarrow (v, v') \end{array} \right\} \\
& \qquad t(v_1, \ldots, v_n) \text{ non-ground}
\end{aligned}$$

Fig. 2. The abstraction rules for pair-independence.

$$\begin{aligned}
\alpha^V_{fr}(x = t) &= \bigcup_{v \in V} \alpha^v_{fr}(x = t) \\
\alpha^v_{fr}(v = x) &= \{vx \Rightarrow v\} \\
\alpha^v_{fr}(v = t(v_1, \ldots, v_n)) &= \{\} \quad t(v_1, \ldots, v_n) \notin \mathcal{V} \\
\alpha^v_{fr}(x = t(v, v_1, \ldots, v_n)) &= \{vx \Rightarrow v\} \\
\alpha^v_{fr}(x = y) &= \{v(x, v)(y, v) \Rightarrow v, vxy \Rightarrow v\} \\
\alpha^v_{fr}(x = t(v_1, \ldots, v_n)) &= \{vx(x, v) \Rightarrow v, v(x, v)(v_1, v) \cdots (v_n, v) \Rightarrow v\} \quad t(v_1, \ldots, v_n) \notin \mathcal{V}\,.
\end{aligned}$$

Fig. 3. The abstraction rules for freeness.

9.2. The abstraction of a set of bindings

Definition 45. The map α^V_{alg} of Definition 43 is extended to H_V as $\alpha^V_{alg}(\exists_W c) = \exists^{Rep_V}_W(\alpha^V_{aux}(c) \cup global_c(\alpha^V_{aux}(c)))$, where

$$\alpha^V_{aux}(\varnothing) = \{(v_1, v_2) \Rightarrow (v_1, v_2), v \Rightarrow v \,|\, \{v, v_1, v_2\} \subseteq V\},$$

$$\alpha^V_{aux}(\{v_1 = t_1, v_2 = t_2, \ldots, v_n = t_n\}) = \alpha^V_{alg}(v_1 = t_1) \bigstar^{IndepFree^1_V} \alpha^V_{aux}(\{v_2 = t_2, \ldots, v_n = t_n\}).$$

The map $global_c$ considers global dependences, i.e., dependences which are lost if we consider each binding in isolation. These dependences are originated by syntactical properties of the constraint c (see Proposition 23). The definition of $global_c : Rep_V \mapsto Rep_V$ is given as

$$global_c(A) = \{global_c(l) \Rightarrow r \,|\, l \Rightarrow r \in A\}$$

for every $A \in Rep_V$, where

$$\begin{aligned}
&global_c(e_1 \cdots e_n) = global_c(e_1) \cdots global_c(e_n)\ e_i \in V \cup V_2 \quad \text{for } i = 1, \ldots, n, \\
&global_c(v) = v, \quad v \in V \\
&global_c((v_1, v_2)) = \begin{cases} v_1 v_2 & \text{if } mgu(c(v_1), c(v_2)) \text{ does not exist} \\ (v_1, v_2) & \text{otherwise.} \end{cases}
\end{aligned}$$

The arrows in $global_c(\alpha^V_{aux}(c))$ are correct for $c \in C_V$.

Lemma 46. *Let $c \in C_V$ and $l \Rightarrow r \in global_c(\alpha^V_{aux}(c))$. Then $c \in \mathbf{l} \to \mathbf{r}$.*

We can state now the final correctness result.

Proposition 47. *Let $h \in H_V$. Then, we have $h \in \gamma_{Rep}\alpha^V_{alg}(h)$ and, therefore, $\alpha_{Rep}\alpha_{IndepFree^1}(\{h\}) \leqslant \alpha^V_{alg}(h)$.*

10. Implementation

In this section we describe a prototypical implementation[5] of the pair-independence and freeness analysis defined in Sections 8 and 9. This implementation has been designed to show

[5] Downloadable at http://www.sci.univr.it/~spoto/ic02_code.tar.gz.

how a domain obtained through linear refinement can be actually used for program analysis. We did not aim at an efficient implementation, though much care has been taken in order to avoid the explosion of the computational cost of the abstract conjunction operator (Definition 30). Note that, since our analysis is goal-independent (Section 3.5), its computational complexity is likely to be greater than that of traditional, goal-dependent analyses, like those based on *Sharing* $\times$ *Free* [28,36]. However, with our approach, once the abstract denotation is computed, it can be instantiated with every input, thus yielding the goal-dependent denotation very quickly. Instead, a goal-dependent analysis requires to re-execute (in an abstract way) the goal in the program.

Our analyser takes a logic program, normalises it, abstracts it, and then computes the fixpoint of the abstract program, by using an abstract version of the s-semantics for computed answers (Section 3.4). Note that other semantics could be used here, like a bottom-up semantics for call-patterns or resultants [20,21]. Indeed, semantics design is totally orthogonal to domain design.

Constraints are manipulated by C procedures, while the normalisation, abstraction, and fixpoint computation phases are written in Prolog. The choice of C as implementation language for the constraints is a consequence of efficiency considerations. Indeed, constraints are represented by arrays of bitmaps. Every basic *token* of information, i.e., the freeness of a variable or the independence of a pair of variables, is associated with a bit position. Elements of $IndepFree_V$ (Definition 24) are then implemented as strings of bits.

We briefly describe normalisation, abstraction, and fixpoint computation. We assume to analyse the following program, which transforms a tree into a heap. We recall that a heap is a tree of natural numbers such that the root of every subtree is the maximum number in the subtree.

```
heapify(void,void).
heapify(tree(X,L,R),Heap):-
    heapify(L,HeapL), heapify(R,HeapR), adjust(X,HeapL,HeapR,Heap).


adjust(X,HeapL,HeapR,tree(X,HeapL,HeapR)):-
    greater(X,HeapL), greater(X,HeapR).
adjust(X,tree(Xl,L,R),HeapR,tree(Xl,HeapL,HeapR)):-
    lt(X,Xl), greater(Xl,HeapR), adjust(X,L,R,HeapL).
adjust(X,HeapL,tree(Xl,L,R),tree(Xl,HeapL,HeapR)):-
    lt(X,Xl), greater(Xl,HeapL), adjust(X,L,R,HeapR).


greater(X,void).
greater(X,tree(Xl,L,R)):-lt(Xl,X).


lt(0,s(_)).
lt(s(X),s(Y)):-lt(X,Y).
```

10.1. Normalisation

The normalisation process transforms a program in such a way that procedure calls are made only in their most general form, with variables `v(0)`, `v(1)`, and so on. Moreover, the logical structure of the program (disjunctions, conjunctions, expansions, and similar) is made apparent, by following Definition 4. Note that this is *not* abstract compilation, which will be considered in the next section. Hence normalisation is independent of the specific domain of analysis.

As an example, the normalisation of the `heapify/2` procedure is

```
heapify(2):-rename(v(16),v(0),rename(v(17),v(1),or(and(
           expand(v(16),bi_eq(v(17),void)),expand(v(17),
           bi_eq(v(16),void))),restrict(v(19),restrict(v(20),
           restrict(v(18),and(expand(v(17),bi_eq(v(16),
           tree(v(18),v(19),v(20)))),expand(v(16),restrict(v(21),
           and(expand(v(20),expand(v(17),expand(v(18),rename(v(0),
           v(19),rename(v(1),v(21),call(heapify(2)))))))),
           expand(v(19),restrict(v(22),and(expand(v(17),expand(v(21),
           expand(v(18),rename(v(0),v(20),rename(v(1),v(22),
           call(heapify(2)))))))),expand(v(20),rename(v(0),v(18),
           rename(v(1),v(21),rename(v(2),v(22),rename(v(3),v(17),
           call(adjust(4)))))))))))))))))).
```

As one can see, the program has been compiled in a code which contains calls to the abstract operations of the domains, as well as built-ins, like `bi_eq`, which unifies two terms, and procedure calls, like `call(adjust(4))`. Note that the arity of the procedures is taken into account.

10.2. Abstraction

The abstraction process substitutes the built-ins with their abstract behaviour, given as a constraint of the abstract domain. Indeed, we are using our domain for abstract compilation and we are not interested in the concrete constraints contained in the concrete program. In the case of `bi_eq` this amounts to applying the abstraction rules of Figs. 1–3. Moreover, this step applies partial evaluation whenever possible, in the sense that if the operand of an abstract operation is known (i.e., it does not contain any procedure call) that operation is applied and substituted in the abstract program with the result.

In the case of the `heapify/2` procedure, we obtain the following abstract program:

```
heapify(2):-rename(v(16),v(0),rename(v(17),v(1),or(abs(327232),
           restrict(v(19),restrict(v(20),restrict(v(18),
           and(abs(327454),expand(v(16),restrict(v(21),
           and(expand(v(20),expand(v(17),expand(v(18),rename(v(0),
           v(19),rename(v(1),v(21),call(heapify(2)))))))),
           expand(v(19),restrict(v(22),and(expand(v(17),
           expand(v(21),expand(v(18),rename(v(0),v(20),
           rename(v(1),v(22),call(heapify(2)))))))),expand(v(20),
```

```
rename(v(0),v(18),rename(v(1),v(21),rename(v(2),v(22),
rename(v(3), v(17), call(adjust(4)))))))))))))))))).
```

An object of the form `abs`(N) represents an abstract element of Rep_V. The number N is the pointer in memory where the constraint, manipulated by C procedures, is stored.

10.3. Fixpoint computation

After the abstraction process, the computation of the fixpoint is just a depth-first evaluation of the abstract program, by using the abstract operations of Section 8.2. The computation is made by exploiting the information contained in the call graph of the program. Namely, if a procedure p is called by a procedure q but not vice versa (even through intermediate procedures), the denotation of p is computed first, and that of q later, by using the denotation already computed for p. In our case, the predicates of the heapifying program are analysed in the following order: `lt`/2, `greater`/2, `adjust`/4, and `heapify`/2.

Even for a program as simple as ours, the abstract analyser does not reach the abstract fixpoint in a reasonable time. This is because the computational cost of the analysis explodes. In order to obtain better performance, we consider two techniques. The first does not introduce any loss of precision, whereas the second, in general, can lead to less precise results.

10.4. Reduction rules

In general, several elements of Rep_V may have the same concretisation through γ_{Rep}. Since the computational complexity of the abstract operators of Section 8.2 is dependent on the dimension of their operands, it is sensible to use those elements of Rep_V which have the smallest dimension. This is accomplished through the use of *reduction rules*, which reduce the dimension of the elements of Rep_V.

Definition 48. A *reduction rule* ρ is a family of maps $\{\rho_V\}_{V \in \wp_f(\mathcal{V})}$ such that

(i) $\rho_V : Rep_V \mapsto Rep_V$,
(ii) $dim(\rho_V(A)) \leqslant dim(A)$ for every $A \in Rep_V$,
(iii) $\gamma_{Rep}(\rho_V(A)) = \gamma_{Rep}(A)$ for every $A \in Rep_V$.

Conditions (ii) and (iii) say that the reduction of $A \in Rep_V$ is an element of smaller dimension but still containing the same information as A. Note that the abstract operators of Section 8.2 are not monotonic w.r.t. *dim*, as the following example shows.

Example 49. The operation *restrict* of Definition 33 is not monotonic w.r.t. the dimension of its argument. For instance, we have

$$restrict_x^{Rep_V}(\{(x,x)(y,y) \Rightarrow (z,z)\}) = \{\},$$
$$restrict_x^{Rep_V}(\{(y,y) \Rightarrow (z,z)\}) = \{(y,y) \Rightarrow (z,z)\}.$$

Therefore, although reduction rules reduce the computational cost of the abstract operations, we do not have any theoretical guarantee that they reduce the computational complexity of the

overall abstract fixpoint computation. However, practical evaluation (Section 11) proves that they are fundamental to improve the efficiency of the analysis.

If we apply a reduction rule after every application of an abstract operator, point (iii) guarantees that we obtain a correct result. However, since this result depends on the representatives chosen for the operands of the operation, we do not have any guarantee, in general, that it is as precise as that obtained by using the abstract operators without any reduction. Note that this would be the case if the abstract operators were optimal, while we just know that they are correct. However, if a reduction rule is reductive w.r.t. the $\preceq$ relation defined below, then using the abstract operators together with a reduction rule cannot lead to a loss of precision, as shown by Proposition 52.

Definition 50. Let $A_1, A_2 \in Rep_V$. We define $A_1 \preceq A_2$ to hold if for every $l_2 \Rightarrow r \in A_2$ there exists $l_1 \Rightarrow r \in A_1$ such that

$$l_1 \subseteq l_2 \cup \{(v, v') \mid v, v' \in V,\ v \neq v' \text{ and } (v, v) \in l_2\}.$$

Definition 51. A *computation* is an element of Rep_V or a term of the form $op(c_1, \ldots, c_n)$, where the c_i's are computations, *op* is the name of one of the abstract operators defined in Section 8.2 and its signature is respected. The evaluation of a computation is defined as

$$[[A]] = A \qquad \text{if } A \in Rep_V,$$
$$[[op(c_1, \ldots, c_n)]] = op([[c_1]], \ldots, [[c_n]]).$$

Moreover, if ρ is a reduction rule, we define

$$[[A]]^\rho = \rho_V(A) \quad \text{if } A \in Rep_V,$$
$$[[op(c_1, \ldots, c_n)]]^\rho = \rho_V(op([[c_1]]^\rho, \ldots, [[c_n]]^\rho)) \quad \text{if } op([[c_1]]^\rho, \ldots, [[c_n]]^\rho) \in Rep_V.$$

Proposition 52. *Given a reduction rule ρ reductive w.r.t. $\preceq$ (i.e., for every $A \in Rep_V$ we have $\rho_V(A) \preceq A$), every computation c is such that $free_V([[c]]) \subseteq free_V([[c]]^\rho)$ and $indep_V([[c]]) \subseteq \mathrm{indep_V}([[\mathrm{c}]]^\rho)$.*

We show now some examples of reduction rules which are reductive w.r.t. $\preceq$ and, therefore, cannot introduce any loss of precision.

The first reduction rule removes entailed arrows.

Proposition 53. *Let $\rho^1 = \{\rho^1_V\}_{V \in \wp_f(\mathcal{V})}$, where*

$$\rho^1_V(A) = \left\{ l \Rightarrow r \in A \;\middle|\; \begin{array}{l} \text{there is } nol' \Rightarrow r \in A \text{ s.t.} \\ l' \subset l \cup \{(v, v') \mid v, v' \in V,\ v \neq v' \text{ and } (v, v) \in l\} \end{array} \right\}$$

for every $A \in Rep_V$. Then ρ^1 is a reduction rule and is reductive w.r.t. $\preceq$. Moreover, it is possible to prove that $\rho^1_V(A) = \bigcap\{X \subseteq A \mid X \preceq A\}$, i.e., $\rho^1(A)$ is the smallest set of A to precede A w.r.t. $\preceq$.

Example 54. Let $V = \{v, x, y, z\}$ and

$$A = \{x(x,y)(x,z)(x,v) \Rightarrow (x,v), x(v,v) \Rightarrow (x,v), xy \Rightarrow y, x(x,v) \Rightarrow (x,v)\}.$$

Then

$$\rho^1_V(A) = \{xy \Rightarrow y, x(x,v) \Rightarrow (x,v)\}.$$

The second reduction rule removes redundant conditions from the left-hand side of the arrows.

Proposition 55. *Let* $\rho^2 = \{\rho^2_V\}_{V \in \wp f(\mathcal{V})}$, *where*

$$\rho^2_V(A) = \{l \setminus \{(v,v') \mid v, v' \in V,\ v \neq v' \text{ and } (v,v) \in l\} \Rightarrow r \mid l \Rightarrow r \in A\}$$

for every $A \in Rep_V$. *Then* ρ^2 *is a reduction rule and is reductive w.r.t.* $\preceq$.

Example 56. Let $V = \{v, x, y, z\}$ and

$$A = \{x(x,y)(y,y)(y,z)(x,z) \Rightarrow (x,z),\ xy \Rightarrow y, y(y,z)(z,z)(v,v) \Rightarrow y\}.$$

Then

$$\rho^2_V(A) = \{x(y,y)(x,z) \Rightarrow (x,z),\ xy \Rightarrow y, y(z,z)(v,v) \Rightarrow y\}.$$

10.5. Improving the efficiency

In it obvious that the efficiency of the analysis can be improved by removing some arrows from the elements of Rep_V. While the reduction rules of the previous section do not introduce any loss of precision, removing arrows can lead to imprecise results in some cases. However, no incorrectness can be introduced. The removal of arrows can be seen as a kind of widening operator [17].

In our implementation we use syntactical equality for the entailment test of Definition 30, which means that some arrows allowed by the theory are not generated by the implementation. Moreover, we have not considered the arrows obtained through the *global* map of Definition 45.

We have not experienced significant loss of precision induced by those choices. Nevertheless, removing arrows is not good practice, and should be applied only when it is strictly necessary. In general, a static analyser could automatically remove some arrows when the computational cost of the fixpoint calculation grows beyond a given threshold. A heuristics must be applied here, in order to select those arrows which do not seem to add too much to the precision of the analysis. It can be dependent on the abstract domain (“choose the longest arrows”) or independent of it (“choose those arrows whose head is a freeness property rather than those whose head is an independence property”).

10.6. The result of the analysis

The analysis of our heapifying program, by using the techniques of Sections 10.4 and 10.5, results in the following output:

```
predicate heapify/2:
?1,(a1,a1),(a0,?1) => ?1
```

```
?l,(al,?l),(a0,a0)=>?l
(?l,?2),(al,al),(a0,?l)=>(?l,?2)
(?l,?2),(al,?l),(a0,a0)=>(?l,?2)
(?l,?l)=>(?l,?l)
(al,al),(a0,?l),a0=>(al,?l)
(a0,a0)=>(al,?l)
(al,al)=>(al,al)
(a0,a0)=>(al,al)
(al,al)=>(a0,?l)
(al,al)=>(a0,a0)
(a0,a0)=>(a0,a0)
(al,al)=>(a0,al)
(a0,a0)=>(a0,al)

predicate adjust/4:
?l,(al,al),(a0,?l),(a3,?l),(a2,a2),(a0,a3)=>?l
?l,(al,?l),(a0,?l),(a0,al),(a3,?l),(a2,a2),(al,a3),(a0,a3),al=>?l
?l,(al,al),(a0,?l),(a3,?l),(a2,?l),(a2,a3),(a0,a3),(a0,a2),a2=>?l
?l,(al,?l),(a0,?l),(a0,al),(a3,?l),(a2,?l),(a2,a3),(al,a3),
(al,a2),(a0,a3),(a0,a2),a2,al=>?l
?l,(al,?l),(a0,?l),(a0,al),(a3,?l),a3,(a2,?l),(a2,a3),
(al,a3),(al,a2),(a0,a3),(a0,a2)=>?l
?l,(al,?l),(a0,?l),(a0,al),(a3,a3),(a2,?l),(al,a2),(a0,a2)=>?l
(?l,?2),(al,al),(a0,?l),(a3,?l),(a2,a2),(a0,a3)=>(?l,?2)
(?l,?2),(al,?l),(a0,?l),(a0,al),(a3,?l),(a2,a2),(al,a3),(a0,a3),
al=>(?l,?2)
(?l,?2),(al,al),(a0,?l),(a3,?l),(a2,?l),(a2,a3),(a0,a3),(a0,a2),
a2=>(?l,?2)
(?l,?2),(al,?l),(a0,?l),(a0,al),(a3,?l),(a2,?l),(a2,a3),(al,a3),
(al,a2),(a0,a3),(a0,a2),a2,al=>(?l,?2)
(?l,?2),(al,?l),(a0,?l),(a0,al),(a3,?l),a3,(a2,?l),(a2,a3),
(al,a3),(al,a2),(a0,a3),(a0,a2)=>(?l,?2)
(?l,?2),(al,?l),(a0,?l),(a0,al),(a3,a3),(a2,?l),(al,a2),
(a0,a2)=>(?l,?2)
(?l,?l)=>(?l,?l)
(a3,a3)=>(al,?l)
(al,?l),(a0,al),(a3,?l),a3,(a2,a3),(al,a3),(al,a2),(a0,a3),
(a0,a2)=>(al,?l)
(a3,a3)=>(al,al)
(al,al)=>(al,al)
(al,?l),(a0,?l),(a0,al),(a3,?l),(a2,?l),(a2,a3),(al,a3),(al,a2),
(a0,a3),(a0,a2)=>(a0,?l)
(a3,a3)=>(a0,?l)
(a0,?l),(a0,al),(a3,?l),a3,(a0,a2)=>(a0,?l)
```

```
(a3,a3) = > (a0,a0)
(a0,a0) = > (a0,a0)
(a3,a3) = > (a0,a1)
(a0,a1),a3,(a0,a2) = > (a0,a1)
(a1,?1),(a0,?1),(a0,a1),(a3,?1),(a2,a2),(a1,a3),(a0,a3),a1 = >
(a3,?1)
(a1,a1),(a0,?1),(a3,?1),(a2,?1),(a2,a3),(a0,a3),(a0,a2),a2 = >
(a3,?1)
(a1,?1),(a0,?1),(a0,a1),(a3,?1),(a2,?1),(a2,a3),(a1,a3),(a1,a2),
(a0,a3),(a0,a2),a2,a1 = > (a3,?1)
(a1,a1),(a0,?1),(a3,?1),(a2,a2),(a0,a3) = > (a3,?1)
(a1,a1),(a0,a0),(a2,a2) = > (a3,?1)
(a1,a1),(a0,a0),(a2,a2) = > (a3,a3)
(a3,a3) = > (a3,a3)
(a0,a1),(a3,?1),a3,(a2,?1),(a2,a3),(a1,a3),(a1,a2),(a0,a3),
(a0,a2) = > (a2,?1)
(a3,a3) = > (a2,?1)
(a2,a2) = > (a2,a2)
(a3,a3) = > (a2,a2)
(a3,a3) = > (a1,a2)
(a0,a1),a3,(a2,a3),(a1,a3),(a1,a2),(a0,a3),(a0,a2) = > (a1,a2)
(a3,a3) = > (a0,a2)
(a0,a1),a3,(a0,a2) = > (a0,a2)

predicate greater/2:
?1,(a1,?1),(a0,?1),(a0,a1) = > ?1
(?1,?2),(a1,?1),(a0,?1),(a0,a1) = > (?1,?2)
(?1,?1) = > (?1,?1)
(a1,?1),(a0,a1) = > (a1,?1)
(a1,a1) = > (a1,a1)
(a0,?1),(a0,a1) = > (a0,?1)
(a0,a0) = > (a0,a0)
(a0,a1) = > (a0,a1)

predicate lt/2:
?1,(a1,?1),(a0,?1),(a0,a1) = > ?1
(?1,?2),(a1,?1),(a0,?1),(a0,a1) = > (?1,?2)
(?1,?1) = > (?1,?1)
(a1,?1),(a0,a1) = > (a1,?1)
(a1,a1) = > (a1,a1)
= > (a0,?1)
= > (a0,a0)
= > (a0,a1)
```

The variable ai is the $(i-1)$th argument position of a procedure, with $i \geqslant 0$.

The analyser concludes that the first argument of `lt/2` is always made ground running that procedure, since its fixpoint contains the arrow $\Rightarrow (a0, a0)$. It concludes that the independence of the two arguments of `greater/2` is maintained by running that procedure, since its fixpoint contains the arrow $(a0, a1) \Rightarrow (a0, a1)$. It concludes that if the fourth argument of `adjoint/4` is ground then the other three arguments are made ground by running that procedure, since its fixpoint contains the arrows $(a3, a3) \Rightarrow (a0, a0)$, $(a3, a3) \Rightarrow (a1, a1)$, and $(a3, a3) \Rightarrow (a2, a2)$. Finally, it concludes that the arrow $(a0, a1), a3, (a0, a2) \Rightarrow (a0, a2)$ holds for `adjust/4`. This arrow says that if the first and the third arguments of `adjust/4` are independent before calling `adjust`, then they are still independent after its execution, provided that the first and the second arguments are independent and the fourth argument is free before the call. The freeness of the fourth argument is necessary. Consider for instance the call `adjust(X, void, tree(O, L, R), tree(Z, void, tree(O, Z, R)))` which results in a computed answer $\exists_{\{w\}}\{\mathtt{X} = \mathtt{L} = \mathtt{Z} = \mathtt{s}(w)\}$, thus making the first and the third arguments of `adjust/4` to share. Instead, the condition $(a0, a1)$ in the left-hand side of the arrow is superfluous. This example shows that our analyser has been able to catch a non-trivial situation when sharing occurs, like that due to the non-freeness of the fourth argument, but its precision can still be improved. Note that we are speaking of the precision *of our analyser*, which must not be confused with the theoretical precision of the *IndepFree*[1] domain.

11. Experimental evaluation

We show now the behaviour of our analyser on some benchmarks. The importance of this evaluation is to show that a goal-independent analysis for freeness and pair-independence like ours can be almost as precise as a goal-dependent analysis, although its cost is definitely higher. We have used SWI-Prolog 3.4.5 over an AMD Duron 600 MHz processor with 128 Mbytes of memory, running Linux 2.2.16. The techniques of Sections 10.4 and 10.5 have been applied.

In Fig. 4, for every benchmark, we report its number of clauses, the maximum number of variables in a clause, the time in seconds spent in the preprocessing phase (normalisation, abstraction, and call graph construction), that spent for the fixpoint computation, and the relative computational cost of the conjunction operation (Definition 30) w.r.t. the other operations of the domain and the shell (preprocessor) which normalises, abstracts, and computes the fixpoint. As one can see, the preprocessing time is always small, while the fixpoint computation is sometimes expensive and seems related to the number of variables in the clauses of the program. When it grows, the time spent for the abstract conjunction explodes, as the fifth column shows. Thus a clever implementation of conjunction is welcome.

We have compared our analyser with a goal-dependent analysis performed by using the *Sharing* × *Free* domain [28] inside the China analyser [3]. The result is shown in Fig. 5. The goal-dependent analysis is definitely more efficient, but must be re-executed for every query. Note that our analysis can be made more efficient through traditional techniques based on *binary decision diagrams* [8]. W.r.t. precision, we have run some abstract queries with the goal-dependent analyser and we have compared the resulting abstract information with what we get by instantiating our goal-independent denotation on the queries. In the third column, "`A`" means "`A` free," while

Benchmark	clauses	vars	Prepr.	Fix.	Conj.	Others	Shell
ackermann.pl	3	4	0.01	0.02	35.0%	7.2%	57.8%
append.pl	2	4	0.01	0.1	77.5%	6.9%	16.5%
eliza.pl	33	6	0.09	0.45	70.1%	6.0%	23.9%
hanoi.pl	4	7	0.02	0.58	88.9%	4.3%	5.7%
heapify.pl	9	6	0.04	20.63	99.5%	0.0%	0.5%
map_coloring.pl	9	5	0.01	0.09	57.6%	12.9%	29.5%
queens.pl	22	5	0.04	0.21	52.7%	14.9%	33.3%
quicksort.pl	11	7	0.03	3.72	97.0%	1.0%	2.0%
openlist.pl	12	8	0.16	0.11	80.3%	10.5%	9.2%

Fig. 4. The analysis times.

Benchmark	Predicate call	Input constraint	Our response	China's
ackermann.pl	ackermann(A,B,C)	ABC(A,B,C)	(A,A)	(A,A)
		(B,B)	(A,A)(B,B)(C,C)	(A,A)(B,B)(C,C)
append.pl	append(A,B,C)	ABC(A,B,C)	B(A,B)	B(A,B)
		BC(A,B,C)	B(A,B)	B(A,B)
		C(A,B,C)	(A,B)	(A,B)
		(C,C)	(A,A)(B,B)(C,C)	(A,A)(B,B)(C,C)
eliza.pl	eliza(A)	A	true	true
hanoi.pl	hanoi(A,B,C,D,E)	ABCDE(A,B,C,D,E)	(A,A)	(A,A)
		(E,E)	(A,A)(B,B) (C,C)(E,E)	(A,A)(B,B) (C,C)(E,E)
heapify.pl	heapify(A,B)	AB(A,B)	true	true
		(A,A)	(A,A)(B,B)	(A,A)(B,B)
map_color.pl	color_map(A,B)	AB(A,B)	true	true
queens.pl	queens(A,B)	AB(A,B)	(A,A)(B,B)	(A,A)(B,B)
		(A,A)	(A,A)(B,B)	(A,A)(B,B)
quicksort.pl	quicksort(A,B)	AB(A,B)	(A,A)(B,B)	(A,A)(B,B)
		(A,A)	(A,A)(B,B)	(A,A)(B,B)
openlist.pl	nil(A,B)	AB(A,B)	B	B
	cons(A,B,C,D,E)	CDE(A,A)(B,D,E) (C,D,E)	CE(A,A)	CE(A,A)
	app(A,B,C,D,E,F)	BDEF(A,C,E,F) (A,D,E,F) (B,C,E,F) (B,D,E,F)	true	DF
	list2open(A,B,C)	BC(B,C)(A,A)	C(A,A)	C(A,A)

Fig. 5. The comparison of our analysis with that done through China.

"(A, B, C)" means that A, B, and C are mutually independent, and is a compact notation for (A, B)(A, C)(B, C). The last two columns show the results of the analysis with our analyser and with China, expressed in our domain. Our analyser is always as precise as China except for app/6, a version of append/3 for incomplete lists. In such a case, the techniques of Section 10.5 have made us lose precision.

12. Conclusions

We have defined a domain for pair-independence and freeness analysis which can be used for abstract compilation. Moreover, we have shown that linear refinement can be applied to a *difficult* case of program analysis.

We have thoroughly described the implementation of all the abstract operators. Therefore, we have been able to implement a static analyser based on our domain. We do not know of any other implementation of a static analysis based on linear refinement. Note that groundness analysis has been *reconstructed* by using linear refinement [37], but it did exist well before this reconstruction. Our implementation is partial, since we have decided to lose precision in order to obtain better performance. However, our evaluation shows that its precision is still comparable with that of a traditional sharing and freeness analysis.

In Section 9, our abstraction map has been designed by considering sets of existential Hebrand constraints $\exists_W c$, where c is an idempotent substitution. The case of non-idempotent substitutions, which corresponds to logic programming without occur-check, has not been considered. In order to do so, we should modify the abstraction map, since some dependences might not be correct anymore for that case. We plan to study how this modification should be done. The correctness of the abstract operators is instead guaranteed.

As suggested in Section 10, the methodology that we have followed for the definition of a representation for the domain $IndepFree^1_V$ and of its abstract operations is almost independent of the abstract property A at hand, being mainly related to the underlying linear refinement technique. Namely, we can imagine an abstract interpretation framework where the abstraction map transforms the existential Herbrand constraints into sets of arrows, in a way dependent on D. The fixpoint engine uses instead abstract operations which are independent of D. It can be hence implemented and optimised as a general purpose static analyser, although specific knowledge about D can allow one to improve its precision and efficiency.

Acknowledgments

We thank Maurice Bruynooghe and Patricia M. Hill for the useful discussions we had with them about the subject of this paper. Thomas Jensen has provided a quiet environment at the *Institut de Recherche en Informatique et Systèmes Aléatoires* of Rennes during the preparation of this paper. Roberto Bagnara has contributed to the experimental evaluation of our analyser. Gianluca Amato has contributed both to the experimental evaluation of our analyser and to the

definition of our *reduction rules*. The anonymous referees provided useful comments which helped us to improve this paper.

Appendix A. Proofs of Section 4

Proof of Proposition 15. Let $\{x,y\} \subseteq V$ with $x \neq y$. We show that the point $(\mathbf{x},\mathbf{y}) \in \mathit{Indep}_V$ is not contained in $\mathit{Indep}_V \rhd \mathit{Indep}_V$. Every point of this last set is the intersection $\bigcap_{i\in I} a_i$, with $I \subseteq \mathbb{N}$, of a set of arrows having their right-hand side in $\{(\mathbf{v_1},\mathbf{v_2}) \,|\, \{v_1,v_2\} \subseteq V\}$, since $\mathbf{l} \to \mathbf{r_1 r_2} = \mathbf{l} \to \mathbf{r_1} \cap \mathbf{l} \to \mathbf{r_2}$. We can assume without any loss of generality that this intersection is minimal, i.e., that no a_i can be left out. We prove that $\bigcap_{i\in I} a_i \neq (\mathbf{x},\mathbf{y})$. Assume by contradiction that $\bigcap_{i\in I} a_i = (\mathbf{x},\mathbf{y})$.

We first prove that every a_i is such that $a_i = \mathbf{l} \to (\mathbf{x},\mathbf{y})$. Assume by contradiction that $a_i = \mathbf{l} \to (\mathbf{v_1},\mathbf{v_2})$, with $v_1 \notin \{x,y\}$. If $\mathbf{l} \subseteq (\mathbf{v_1},\mathbf{v_1})$ or $\mathbf{l} \subseteq (\mathbf{v_2},\mathbf{v_2})$ then a_i is tautological and can be left out. By the hypothesis on a_i, we can assume $\mathbf{l} \not\subseteq (\mathbf{v_1},\mathbf{v_1})$ and $\mathbf{l} \not\subseteq (\mathbf{v_2},\mathbf{v_2})$. Let $h = \{v_1 = v_2\}$. We have $h \in (\mathbf{x},\mathbf{y})$ by the hypothesis $v_1 \notin \{x,y\}$. Let $h' = \{v = \mathtt{a} \,|\, \mathbf{l} \subseteq (\mathbf{v},\mathbf{v})\} \in \mathbf{l}$. The constraint h' makes ground exactly those variables that l requires to be ground. Since v_1 and v_2 are unbound in h', we have $h \bigstar^{H_V} h' = h' \cup \{v_1 = v_2\} \notin (\mathbf{v_1},\mathbf{v_2})$. Since $h' \in \mathbf{l}$, we have $h \notin \mathbf{a_i}$. But $h \in (\mathbf{x},\mathbf{y})$. This proves that the right-hand side of a_i must be $(\mathbf{x},\mathbf{y})$.

We prove now that every non-tautological arrow $a_i = \mathbf{l} \to (\mathbf{x},\mathbf{y})$ with $(\mathbf{x},\mathbf{y}) \subseteq a_i$ is such that $\mathbf{l} \subseteq (\mathbf{v},\mathbf{v})$ for every $v \in V \setminus \{x,y\}$. Indeed, assume $\mathbf{l} \not\subseteq (\mathbf{v},\mathbf{v})$ with $v \in V \setminus \{x,y\}$. If $\mathbf{l} \subseteq (\mathbf{x},\mathbf{x})$ or $\mathbf{l} \subseteq (\mathbf{y},\mathbf{y})$ then a_i would be tautological. Therefore, we can assume $\mathbf{l} \not\subseteq (\mathbf{x},\mathbf{x})$ and $\mathbf{l} \not\subseteq (\mathbf{y},\mathbf{y})$. Consider $h = \{v = \mathtt{f}(x,y)\} \in (\mathbf{x},\mathbf{y})$ and $h' = \exists_{\{w\}}(\{v = \mathtt{f}(w,w)\} \cup \{v' = \mathtt{a} \,|\, \mathbf{l} \subseteq (\mathbf{v}',\mathbf{v}')\})$. We have $h' \in \mathbf{l}$ by the hypothesis on v. Since $h \bigstar^{H_V} h' = \{v = \mathtt{f}(x,x), x = y\} \cup \{v' = \mathtt{a} \,|\, \mathbf{l} \subseteq (\mathbf{v}',\mathbf{v}')\}$, we have $h \bigstar^{H_V} h' \notin (\mathbf{x},\mathbf{y})$. This shows that $h \notin a_i$, a contradiction since $(\mathbf{x},\mathbf{y}) \subseteq a_i$ and $h \in (\mathbf{x},\mathbf{y})$.

In conclusion, every a_i must have the form $\mathbf{l} \to (\mathbf{x},\mathbf{y})$ with $\mathbf{l} \subseteq (\mathbf{v},\mathbf{v})$ for every $v \in V \setminus \{x,y\}$. Moreover, we can assume $\mathbf{l} \not\subseteq (\mathbf{x},\mathbf{x})$ and $\mathbf{l} \not\subseteq (\mathbf{y},\mathbf{y})$, otherwise a_i would be tautological. Let $v \in V \setminus \{x,y\}$. We can find such a v since $\#V \geqslant 3$. Let $h = \{v = x, y = x\} \notin (\mathbf{x},\mathbf{y})$. Given $h' \in \mathbf{l}$, since v is ground in h', we conclude that if $h \bigstar^{H_V} h'$ exists then $h \bigstar^{H_V} h' \in (\mathbf{x},\mathbf{y})$. Therefore, we have $h \in \bigcap_{i\in I} a_i$. This proves that $\bigcap_{i\in I} a_i \neq (\mathbf{x},\mathbf{y})$. □

Proof of Example 14. Let by contradiction $\alpha(h_1) \bigstar^{\mathit{Indep}^1_V} \alpha(h_2) \subseteq (\mathbf{y},\mathbf{z})$. By definition of $\to$, this means that $\alpha(h_1) \subseteq \alpha(h_2) \to (\mathbf{y},\mathbf{z})$. Hence every $h \in \alpha(h_2)$ is such that its concrete conjunction with h_1 does not make y and z to share. By considering the enumeration of all possible arrows, we conclude that $\alpha(h_2) = \cap A$, where

$$A = \left\{ \begin{array}{l} (\mathbf{v},\mathbf{y}),\ (\mathbf{v},\mathbf{z}),\ (\mathbf{x},\mathbf{y}),\ (\mathbf{x},\mathbf{z}),\ (\mathbf{y},\mathbf{z}) \\ (\mathbf{x},\mathbf{x}) \to (\mathbf{v},\mathbf{v}),\ (\mathbf{v},\mathbf{v}) \to (\mathbf{x},\mathbf{x}),\ (\mathbf{v},\mathbf{y})(\mathbf{x},\mathbf{y}) \to (\mathbf{v},\mathbf{y}) \\ (\mathbf{x},\mathbf{x}) \to (\mathbf{v},\mathbf{y}),\ (\mathbf{v},\mathbf{z})(\mathbf{x},\mathbf{z}) \to (\mathbf{v},\mathbf{z}),\ (\mathbf{x},\mathbf{y})(\mathbf{v},\mathbf{y}) \to (\mathbf{x},\mathbf{y}) \\ (\mathbf{x},\mathbf{x}) \to (\mathbf{v},\mathbf{z}),\ (\mathbf{x},\mathbf{z})(\mathbf{v},\mathbf{z}) \to (\mathbf{x},\mathbf{z}),\ (\mathbf{y},\mathbf{z})(\mathbf{v},\mathbf{v}) \to (\mathbf{y},\mathbf{z}) \\ (\mathbf{v},\mathbf{v}) \to (\mathbf{x},\mathbf{y}),\ (\mathbf{v},\mathbf{v}) \to (\mathbf{x},\mathbf{z}),\ (\mathbf{y},\mathbf{z})(\mathbf{x},\mathbf{y})(\mathbf{v},\mathbf{y}) \to (\mathbf{y},\mathbf{z}) \\ (\mathbf{y},\mathbf{z})(\mathbf{x},\mathbf{x}) \to (\mathbf{y},\mathbf{z}),\ (\mathbf{y},\mathbf{z})(\mathbf{x},\mathbf{z})(\mathbf{v},\mathbf{z}) \to (\mathbf{y},\mathbf{z}) \end{array} \right\}.$$

Hence $h = \{x = \mathtt{f}(v, v)\} \in \alpha(h_2)$, a contradiction since the conjunction of h and h_1 makes y and z to share. □

Proof of Proposition 20. Let $x \in V$. We show that $\mathbf{x} \in \mathit{Free}_V$ is not contained in $\mathit{Free}_V \triangleright \mathit{Free}_V$. Every element of this last set is the intersection $\bigcap_{i\in I} a_i$, with $I \subseteq \mathbb{N}$, of a set of arrows having their right-hand side in $\{v \mid v \in V\}$. We can assume without any loss of generality that this intersection is minimal, i.e., that no a_i can be left out. We prove that $\bigcap_{i\in I} a_i \neq \mathbf{x}$. Assume by contradiction that $\bigcap_{i\in I} a_i = \mathbf{x}$.

We first prove that there is no a_i such that $a_i = \mathbf{l} \to \mathbf{v}$ with $v \neq x$. Indeed, in such a case we would have $h = \{v = \mathtt{a}\} \notin a_i$, since $\varnothing \in \mathbf{l}$ and $\varnothing \bigstar^{H_V} h = h \notin v$. But $h \in \mathbf{x}$, which is a contradiction since $\mathbf{x} \subseteq a_i$. Therefore, every a_i must have the form $\mathbf{l} \to \mathbf{x}$. Since $\#V \geqslant 2$, there exists $y \in V$, $y \neq x$. Consider $h = \{y = \mathtt{a}\} \in \mathbf{x}$. We have $h \notin a_i$ since $\{y = x\} \in \mathbf{l}$ and $\{y = x\} \bigstar^{H_V} h = \{x = \mathtt{a}, y = \mathtt{a}\} \notin \mathbf{x}$. Then $\mathbf{x} \not\subseteq a_i$. This proves that I is empty, i.e., $\bigcap_{i\in I} a_i = H_V \neq \mathbf{x}$, a contradiction. □

Proof of Example 19. The constraint h_1 belongs to $\mathbf{yz}$. We want to show that $\mathbf{yz}$ is the abstraction of h_1 in Free_V^1. Indeed, if $h_1 \in \mathbf{l} \to \mathbf{r}$ and $\mathbf{l} \to \mathbf{r} \neq H_V$, then $\mathbf{r} \neq H_V$. Since $h_3 = \{x = y, z = y\}$ belongs to every element of Free_V and therefore to $\mathbf{l}$, we would have $h_3 \bigstar^{H_V} h_1 \in \mathbf{r}$, which is a contradiction, since $\mathbf{r} \neq H_V$ and all three variables are made non-free by the conjunction. Then h_1 is abstracted to $\mathbf{yz}$ in Free_V^1. By symmetry, h_2 is abstracted to $\mathbf{xz}$ in Free_V^1. The same argument used in Example 18 in the case of Free_V shows that the best correct approximation of the $\bigstar^{\wp(H_V)}$ operation in not contained in $\mathbf{z}$ when applied to $\alpha(h_1)$ and $\alpha(h_2)$. □

Appendix B. Proofs of Section 8

Proof of Proposition 29. Let $l \Rightarrow v \in A$, $\exists_W c \in \mathbf{l} \to \mathbf{v}$, $\theta = \{v'' = \mathtt{a} \mid v'' \in V \cup W\}$ and $h = \{v'' = c(v'')\theta \mid v'' \in V,\ (v'', v'') \in l\}$. If $v'(v', v') \not\subseteq l$ for any $v' \in V$ then $h \in \mathbf{l}$. Hence $h \bigstar^{H_V} \exists_W c \in \mathbf{v}$ and, since v is upward closed, $\exists_W c \in \mathbf{v}$. Thus $\exists_W c \in \cap\{\mathbf{v} \mid l \Rightarrow \mathbf{v} \in A \text{ and } v'(v', v') \not\subseteq l \text{ for any } v' \in V\} = \mathbf{free_V}(\mathbf{A})$.

The result for pair-independence can be proved similarly by using $h = \varnothing$. □

Proof of Proposition 32. Let $A_1, A_2 \in \mathit{Rep}_V$. Since $\gamma_{\mathit{IndepFree}^1}$ is the identity map, it suffices to prove that

$$\gamma_{\mathit{Rep}}(A_1) \bigstar^{\wp(H_V)} \gamma_{\mathit{Rep}}(A_2) \subseteq \gamma_{\mathit{Rep}}(A_1 \bigstar^{\mathit{Rep}_V} A_2),$$

i.e., for every $h_1 \in \gamma_{\mathit{Rep}}(A_1)$ and $h_2 \in \gamma_{\mathit{Rep}}(A_2)$ we have $h_1 \bigstar^{H_V} h_2 \in \gamma_{\mathit{Rep}}(A_1 \bigstar^{\mathit{Rep}_V} A_2)$. In turn, this means that we have to prove that every arrow $l_1 \cdots l_n \Rightarrow r \in A_1 \bigstar^{\mathit{Rep}_V} A_2$ is such that $h_1 \bigstar^{H_V} h_2 \in \mathbf{l}_1 \cdots \mathbf{l}_n \to \mathbf{r}$. We have two cases. The first case is when $h_2 \in \mathbf{r_1} \cdots \mathbf{r_n} \to \mathbf{r}$ and $h_1 \in \mathbf{l_i} \to \mathbf{r'_i}$ with $\mathbf{r'_i} \subseteq \mathbf{r_i}$ for every $i = 1, \ldots, n$. Let $h \in \mathbf{l}_1 \cdots \mathbf{l}_n$. We have $h \bigstar^{H_V} (h_1 \bigstar^{H_V} h_2) = (h \bigstar^{H_V} h_1) \bigstar^{H_V} h_2$. Since $h \in \mathbf{l_i}$ and $h_1 \in \mathbf{l_i} \to \mathbf{r'_i}$ for every $i = 1, \ldots, n$, we have $h \bigstar^{H_V} h_1 \in \mathbf{r}'_1 \cdots \mathbf{r}'_n$, and since $h_2 \in \mathbf{r_1} \cdots \mathbf{r_n} \to \mathbf{r}$ and $\mathbf{r}'_1 \cdots \mathbf{r}'_n \subseteq \mathbf{r}_1 \cdots \mathbf{r}_n$ we conclude that $(h \bigstar^{H_V} h_1) \bigstar^{H_V} h_2 \in \mathbf{r}$. The second case in the definition of $\bigstar^{\mathit{Rep}_V}$ can be proved symmetrically.

A naive implementation of $\bigstar^{\mathit{Rep}_V}$ considers every unfolding of an arrow of $A_1 \cup T$ with the arrows of $A_2 \cup T$ and vice versa. Thus, its worst-case time complexity is $O((a_1 + t)(a_2 + t)k)$. □

Proof of Proposition 35. We have to prove that
(i)

$$restrict_x^{\wp(H_V)}(\gamma_{Rep_V}(A)) \subseteq \gamma_{Rep_{V\setminus x}}(restrict_x^{Rep_V}(A)).$$

(ii)

$$rename_{x\to n}^{\wp(H_V)}(\gamma_{Rep_V}(A)) \subseteq \gamma_{Rep_{(V\setminus x)\cup n}}(rename_{x\to n}^{Rep_V}(A)).$$

(i) The result follows by proving that given $\exists_W c \in \gamma_{Rep_V}(A)$ and $l \Rightarrow r \in A$ such that $(x,x) \notin l$, $r \not\equiv x$ and $r \not\equiv (x,v)$ for every $v \in V$, we have $\exists_{W\cup N} c[x \mapsto N] \in \mathbf{l}'_{V\setminus x} \to \mathbf{r}_{V\setminus x}$, where $l' = l \setminus X$. Let $\exists_{W'} c' \in \mathbf{l}'_{V\setminus x}$. We have $\exists_{W'} c' \in \mathbf{l}'$. Since $x \notin dom(c') \cup rng(c')$ we have $\exists_{W'} c' \in \mathbf{l}$. From $\exists_W c \in \mathbf{l} \to \mathbf{r}$, we conclude that $\exists_{W'} c' \bigstar^{H_V} \exists_W c \in \mathbf{r}$. Moreover, we have $\exists_{W'} c' \bigstar^{H_V} \exists_{W\cup N} c[x \mapsto N] = \exists_{W'\cup W\cup N} mgu(c' \cup c[x \mapsto N])$. From $c' \in C_{V\setminus x}$ we conclude that $mgu(c' \cup c[x \mapsto N])(v) = mgu(c' \cup c)(v)[x \mapsto N]$ for every $v \in V \setminus x$. Since $r \not\equiv x$ and $r \not\equiv (v,x)$ for every $v \in V$, from $\exists_{W'} c' \bigstar^{H_V} \exists_W c = \exists_{W'\cup W} mgu(c' \cup c) \in \mathbf{r}$ we conclude that $\exists_{W'} c' \bigstar^{H_V} \exists_{W\cup N} c[x \mapsto N] \in \mathbf{r}$. Finally, since $x \notin dom(mgu(c' \cup c[x \mapsto N])) \cup rng(mgu(c' \cup c[x \mapsto N]))$, we conclude that $\exists_{W'} c' \bigstar^{H_{V\setminus x}} \exists_{W\cup N} c[x \mapsto N] \in \mathbf{r}_{V\setminus x}$.

(ii) Let $\exists_W c \in \gamma_{Rep_V}(A)$. We prove that $rename_{x\to n}^{H_V}(h) \in \mathbf{l}_{(V\setminus x)\cup n} \to \mathbf{r}_{(V\setminus x)\cup n}$ for every $l \Rightarrow r \in A[x \mapsto n]$, i.e., $\exists_W c[x \mapsto n] \in \mathbf{l}[\mathbf{x} \mapsto \mathbf{n}]_{V\setminus x\cup n} \to \mathbf{r}[\mathbf{x} \mapsto \mathbf{n}]_{(V\setminus x)\cup n}$ for every $l \Rightarrow r \in A$. Let $\exists_{W'} c' \in \mathbf{l}[\mathbf{x} \mapsto \mathbf{n}]_{(V\setminus x)\cup n}$. Since $c' \in C_{(V\setminus x)\cup n}$, we conclude that $\exists_{W'} c'[n \mapsto x] \in \mathbf{l}$. Thus, we have $\exists_{W'} c'[n \mapsto x] \bigstar^{H_V} \exists_W c = \exists_{W'\cup W} mgu(c' \ [n \mapsto x] \cup c) \in \mathbf{r}$ since $\exists_W c \in \mathbf{l} \to \mathbf{r}$, and $\exists_{W'\cup W} mgu(c'[n \ \mapsto x] \cup c) = \exists_{W'\cup W} mgu((c' \cup c[x \mapsto n])[n \mapsto x]) = (\exists_{W'} c' \bigstar^{H_{(V\setminus x)\cup n}} \exists_W c[x \mapsto n])[n \mapsto x] \in \mathbf{r}$. Since $c' \in C_{(V\setminus x)\cup n}$, $\exists_{W'} c' \bigstar^{H_{(V\setminus x)\cup n}} \exists_W c[x \mapsto n] \in \mathbf{r}[\mathbf{x} \mapsto \mathbf{n}]_{(V\setminus x)\cup n}$. Hence $\exists_W c[x \mapsto n] \in \mathbf{l}[\mathbf{x} \mapsto \mathbf{n}]_{(V\setminus x)\cup n} \to \mathbf{r}[\mathbf{x} \mapsto \mathbf{n}]_{(V\setminus x)\cup n}$.

For the result about complexity, a naive implementation of $restrict_n^{Rep_V}(A)$ considers every arrow of A and scans its body to check whether it must be removed or not. Hence its worst-case time complexity is linear in the dimension of A. Similarly for $rename_{x\to n}^{Rep_V}$. □

Proof of Lemma 36. Let $h = \exists_W c \in \mathbf{l}_V \to \mathbf{r}_V$ and let $\exists_{W'} c' \in \mathbf{l}_{V\cup x}$ and $N \in \mathcal{W}$ fresh. We have $\exists_{W'\cup N} c'|_V[x \mapsto N] \in \mathbf{l}$ because x does not occur in l. Since $\exists_{W'\cup N} c'|_V[x \mapsto N] \bigstar^{H_V} \exists_W c = \exists_{W'\cup W\cup N} mgu(c'|_V[x \mapsto N] \cup c) \in \mathbf{r}$ and $c \in C_V$ with $x \notin V$, we conclude that $\exists_{W'\cup W} mgu(c'|_V \cup c) \in \mathbf{r}_{V\cup x}$ and since x does not occur in r we conclude that $\exists_{W'\cup W} mgu(c' \cup c) \in \mathbf{r}_{V\cup x}$, i.e., $\exists_{W'} c' \bigstar^{\wp(H_{V\cup x})} \exists_W c \in \mathbf{r}_{V\cup x}$. □

Proof of Lemma 37. Let $\exists_{W'} c' \in \mathbf{l}[\mathbf{v_2} \mapsto \mathbf{v_1}, \mathbf{v_1} \mapsto \mathbf{v_2}]$ and $c'' \in C_V$ such that $c'' = \{x[v_2 \mapsto v_1, v_1 \mapsto v_2] = t[v_2 \mapsto v_1, v_1 \mapsto v_2] \,|\, x = t \in c'\}$. By definition, we have $\exists_{W'} c'' \in \mathbf{l}$. Hence $\exists_{W'\cup W} mgu(c'' \cup c) = \exists_{W'} c'' \bigstar^{H_V} \exists_W c \in \mathbf{r}$. Since $\{v_1, v_2\} \cap (dom(c) \cup rng(c)) = \varnothing$, we have $mgu(c'' \cup c) = \{x[v_2 \mapsto v_1, \ v_1 \mapsto v_2] = t[v_2 \mapsto v_1, v_1 \mapsto v_2] \,|\, x = t \in mgu(c' \cup c)\}$ and $\exists_{W'} c' \bigstar^{H_V} \exists_W c = \exists_{W'\cup W} mgu(c' \cup c) \in \mathbf{r}[v_2 \mapsto v_1, v_1 \mapsto v_2]$. The converse holds by symmetry. □

Proof of Proposition 40. Let $x \in \mathcal{V} \setminus V$, $\{?_1, ?_2\} \subseteq V$, and $A \in Rep_V$. We have to prove that

$$expand_x^{\wp(H_V)}(\gamma_{Rep_V}(A) \cap X) \subseteq \gamma_{Rep_{V\cup x}}(expand_x^{Rep_V}(A)),$$

where $X = \{\exists_W c \in H_V \,|\, \{?_1, ?_2\} \cap (dom(c) \cup rng(c)) = \varnothing\}$. By Lemma 36, the arrows in A correctly approximate $expand_x^{\wp(H_V)}(\gamma_{Rep_V}(A))$. Moreover, given a constraint $\exists_W c \in expand_x^{\wp(H_V)}$

$(\gamma_{Rep_V}(A) \cap X)$, since $\{?_1, x\} \cap \{dom(c) \cup rng(c)\} = \varnothing$ and x does not occur in $l \Rightarrow r$ for the hypothesis $x \notin V$, by Lemma 37 we conclude that $h \in \mathbf{l}[?_\mathbf{1} \mapsto \mathbf{x}] \to \mathbf{r}[?_\mathbf{1} \mapsto \mathbf{x}]$ for every $l \Rightarrow r \in A$. Moreover, since $\{?_1, ?_2\} \cap \{dom(c) \cup rng(c)\} = \varnothing$ and $?_1$ does not occur in $l[?_1 \mapsto x] \Rightarrow r[?_1 \mapsto x]$, by Lemma 37 again we conclude that $h \in \mathbf{l}[?_\mathbf{1} \mapsto \mathbf{x}][?_\mathbf{2} \mapsto ?_\mathbf{1}] \to \mathbf{r}[?_\mathbf{1} \mapsto \mathbf{x}][?_\mathbf{2} \mapsto ?_\mathbf{1}]$ for every $l \Rightarrow r \in A$.

For the result about complexity, a naive implementation of $expand_x^{Rep_V}(A)$ scans the arrows of A in order to incarnate $?_1$ into x and $?_2$ into $?_1$. Therefore, its worst-case time complexity is bounded by the dimension of A. □

Proof of Proposition 42. Let $A_1, A_2 \in Rep_V$. We have to prove that

$$\gamma_{Rep}(A_1) \cup \gamma_{Rep}(A_2) \subseteq \gamma_{Rep}(A_1 \cup^{Rep_V} A_2).$$

For every $l \Rightarrow r \in A_1 \cup^{Rep_V} A_2$, we have $l = l_1 l_2$ with $l_1 \Rightarrow r \in A_1$ and $l_2 \Rightarrow r \in A_2$. Then $\gamma_{Rep}(A_1) \subseteq \gamma_{Rep}(l_1 \Rightarrow r) \subseteq \gamma_{Rep}(l_1 l_2 \Rightarrow r)$ and $\gamma_{Rep}(A_2) \subseteq \gamma_{Rep}(l_2 \Rightarrow r) \subseteq \gamma_{Rep}(l_1 l_2 \Rightarrow r)$. Since this holds for every $l \Rightarrow r \in A_1 \cup^{Rep_V} A_2$, we have the thesis.

For the result about complexity, a naive algorithm for $\cup^{Rep_V}$ scans the heads of the arrows in A_1 for elements which belong to the head of some arrow in A_2 and merges their bodies when this happens. Therefore, its complexity is $O(a_1 a_2)$. □

Appendix C. Proofs of Section 9

Proof of Lemma 44. It suffices to prove that every $l \Rightarrow r \in \alpha_{gr}^V(z = t) \cup \alpha_{indep}^V(z = t) \cup \alpha_{fr}^V(z = t)$ is such that $\{z = t\} \in \mathbf{l} \to \mathbf{r}$. This is true if $l \Rightarrow r \in \alpha_{gr}^V(z = t)$, since $(\mathbf{v}, \mathbf{v}) = \{\exists_W c \in H_V \mid c(v) \in terms(\Sigma, \varnothing)\}$ and $\alpha_{gr}^V(z = t)$ contains obvious groundness dependences. For α_{indep}^V and α_{fr}^V, given $h = \exists_W c \in \mathbf{l}$, we have $h \bigstar^{H_V} \{z = t\} = \exists_W mgu(c\ mgu(c(z) = tc))$, if it is defined. Let $c' = mgu(c(z) = tc)$. Since $dom(c') \cap dom(c) = \varnothing$, we have $h \bigstar^{H_V} \{z = t\} = \exists_W (cc')$, if it exists. Assume it exists. We consider every single arrow contained in $\alpha_{indep}(z = t) \cup \alpha_{fr}(z = t)$ and we prove that $h' = \exists_W (cc') \in \mathbf{r}$. If $t(v_1, \ldots, v_n)$ is a term, we write $t(t_1, \ldots, t_n)$ for $t(v_1, \ldots, v_n)\ [v_1 \mapsto t_1, \ldots v_n \mapsto t_n]$.

Consider $x(v, v') \Rightarrow (v, v') \in \alpha_{indep}^V(x = t(v, v', v_1, \ldots, v_n))$, with $n \geqslant 0$. Since $c(x) \in V$, we have $c' = \{c(x) = t(c(v), c(v'), c(v_1), \ldots, c(v_n))\}$. Note that $c(x) \notin vars(c(v)) \cup vars(c(v'))$, since otherwise c' and h' would not exist. Therefore, we have $(cc')(v) = c(v)c' = c(v)$ and $(cc')(v') = c(v')c' = c(v')$, and from $h \in (\mathbf{v}, \mathbf{v}')$ we conclude that $vars((cc')(v)) \cap vars\ ((cc')(v')) = \varnothing$. Hence $h' \in (\mathbf{v}, \mathbf{v}')$.

Consider $(v', v)(v', v_1) \cdots (v', v_n) \Rightarrow (v', v) \in \alpha_{indep}^V(v = t(v_1, \ldots, v_n))$, with $n \geqslant 1$. We have $c' = mgu\{c(v) = t(c(v_1), \ldots, c(v_n))\}$, with $dom(c') \cup rng(c') \subseteq vars(c(v)) \cup \bigcup_{i=1,\ldots,n} vars(c(v_i))$ and from the fact that $h \in (\mathbf{v}', \mathbf{v})(\mathbf{v}', \mathbf{v_1}) \cdots (\mathbf{v}', \mathbf{v_n})$ we have $dom(c') \cap vars(c(v')) = rng(c') \cap vars(c(v')) = \varnothing$. Therefore, we have $(cc')(v) = c(v)c'$, $(cc')(v') = c(v')c' = c(v')$ and $vars((cc')(v)) \cap vars((cc')(v')) = vars(c(v)c') \cap vars(c(v')) \subseteq (vars(c(v)) \cup rng(c')) \cap vars\ (c(v')) = rng(c') \cap vars(c(v')) = \varnothing$. Hence $h' \in (\mathbf{v}, \mathbf{v}')$.

Consider $(v', v)(v', x)(v', v_1) \cdots (v', v_n) \Rightarrow (v', v) \in \alpha_{indep}^V(x = t(v, v_1, \ldots, v_n))$. We have $c' = mgu\{c(x) = t(c(v), c(v_1), \ldots, c(v_n))\}$ and $dom(c') \cup rng(c') \subseteq vars(c(x)) \cup vars(c(v)) \cup \bigcup_{i=1,\ldots,n} vars(c(v_i))$ and since $h \in (\mathbf{v}', \mathbf{v})(\mathbf{v}', \mathbf{x})(\mathbf{v}', \mathbf{v_1}) \cdots (\mathbf{v}', \mathbf{v_n})$ we conclude that $dom(c') \cap vars(c(v')) =$

$rng(c') \cap vars(c(v')) = \varnothing$. Therefore, we have $(cc')(v) = c(v)c'$, $(cc')(v') = c(v')c' = c(v')$ and $vars((cc')(v)) \cap vars((cc')(v')) = vars\ \ (c(v)c') \cap vars(c(v')) \subseteq (vars(c(v)) \cup rng(c')) \cap vars(c(v')) = rng(c') \cap vars(c(v')) = \varnothing$. Hence $h' \in (\mathbf{v}, \mathbf{v}')$.

Consider $(v', v)(v', x)x \Rightarrow (v', v) \in \alpha^V_{indep}(x = t(v, v_1, \ldots, v_n))$. Since $c(x) \in V$, we have $c' = \{c(x) = t(c(v), c(v_1), \ldots, c(v_n))\}$. Since $c(x) \neq v'$, we have $cc'(v') = c(v')$. Moreover, $cc'(v) = c(v)$ if $c(x) \neq v$ and $cc'(v) = t(c(v), c(v_1), \ldots, c(v_n))$ otherwise. However, in the second case, from $c(x) = v$ and since $h' = h\bigstar^{H_V}\{x = t(v, v_1, \ldots, v_n)\}$ is defined, we conclude that $t(v, v_1, \ldots, v_n)$ is syntactically equal to v. Then, even in the second case, we have $cc'(v) = c(v)$. Since $vars(c(v)) \cap vars(c(v')) = \varnothing$, we conclude that $vars(cc'(v)) \cap vars(cc'(v')) = \varnothing$, i.e., $h' \in (v, v')$.

Consider $(v, v') \Rightarrow (v, v') \in \alpha^V_{indep}(x = t)$, with t ground. We have that $c' = mgu\{c(x) = t\}$ is such that $c'(y)$ is ground for every $y \in dom(c')$. Therefore, we have that $vars((cc')(v)) \cap vars((cc')(v')) = vars(c(v)c') \cap vars(c(v')c') \subseteq vars(c(v)) \cap vars(c(v')) = \varnothing$ since $h \in (v, v')$. Hence $h' \in (v, v')$.

Consider $(v, v')(v', x)(v', v_1) \cdots (v', v_n) \Rightarrow (v, v') \in \alpha^V_{indep}(x = t(v_1, \ldots, v_n))$ with $n \geqslant 1$. We have $c' = mgu\{c(x) = t(c(v_1), \ldots, c(v_n))\}$, with $dom(c') \cup rng(c') \subseteq vars(c(x)) \cup \bigcup_{i=1,\ldots,n} vars(c(v_i))$ and since $h \in (\mathbf{v}, \mathbf{v}')(\mathbf{v}', \mathbf{x})(\mathbf{v}', \mathbf{v_1}) \cdots (\mathbf{v}', \mathbf{v_n})$ we conclude that $dom(c') \cap vars(c(v')) = rng(c') \cap vars(c(v')) = \varnothing$. Therefore, we have $(cc')(v) = c(v)c'$ and $(cc')(v') = c(v')c' = c(v')$. Then $vars((cc')(v)) \cap vars((cc')(v')) = vars(c(v)c') \cap vars(c(v')) \subseteq (vars(c(v)) \cup rng(c')) \cap vars(c(v')) = rng(c') \cap vars(c(v')) = \varnothing$, and $h' \in (\mathbf{v}, \mathbf{v}')$. The same proof holds for its symmetrical arrow.

Consider $(v, v')(v, x)(v', x)x \Rightarrow (v, v') \in \alpha^V_{indep}(x = t(v_1, \ldots, v_n))$. Since $c(x) \in V$, we have $c' = \{c(x) = t(c(v_1), \ldots, c(v_n))\}$. From $h \in (\mathbf{v}, \mathbf{x})(\mathbf{v}', \mathbf{x})$ we have $c(x) \neq v$ and $c(x) \neq v'$. Then $cc'(v) = c(v)$ and $cc'(v') = c(v')$. From $h \in (v, v')$, we conclude that $vars(cc'(v)) \cap vars\ (cc'(v')) = \varnothing$, i.e., $h' \in (\mathbf{v}, \mathbf{v}')$.

Consider $vx \Rightarrow v \in \alpha^V_{fr}(v = x)$. We have $c' = \{c(v) = c(x)\}$. Since $\{c(v), c(x)\} \subseteq V$, we have $(cc')(v) = c(v)c' = c(x) \in V$ and $h' \in \mathbf{v}$.

Consider $vx \Rightarrow v \in \alpha^V_{fr}(x = t(v, v_1, \ldots, v_n))$, with $n \geqslant 0$. Since $c(x) \in V$, we have $c' = \{c(x) = t(c(v), c(v_1), \ldots, c(v_n))\}$. If $c(v) \neq c(x)$, we have $(cc')(v) = c(v)c' = c(v) \in V$. Otherwise, c' exists only if $t(v, v_1, \ldots, v_n) \equiv v$. In this case, we have $(cc')(v) = c(v)c' = v \in V$. In both cases we have $(cc')(v) \in V$ and $h' \in \mathbf{v}$.

Consider $v(x, v)(y, v) \Rightarrow v \in \alpha^V_{fr}(x = y)$. We have $c' = mgu\{c(x) = c(y)\}$. Since $c(v) \in V$ and $dom(c') \subseteq vars(c(x)) \cup vars(c(y))$, from $h \in (\mathbf{x}, \mathbf{v})(\mathbf{y}, \mathbf{v})$ we conclude that $c(v) \notin dom(c')$. Therefore, $(cc')(v) = c(v)c' = c(v) \in V$ and $h \in v$.

Consider $vxy \Rightarrow v \in \alpha^V_{fr}(x = y)$. We have $c' = \{c(x) = c(y)\}$ and since $\{c(v), c(x), c(y)\} \subseteq V$ we conclude that $(cc')(v) = c(v)c' \in V$. Therefore, $h' \in \mathbf{v}$.

Consider $v(x, v)(v_1, v) \cdots (v_n, v) \Rightarrow v \in \alpha^V_{fr}(x = t(v_1, \ldots, v_n))$, with $n \geqslant 0$ and $t(v_1, \ldots, v_n) \notin \mathcal{V}$. We have $c' = mgu\{c(x) = t(c(v_1), \ldots, c(v_n))\}$. Since $c(v) \in V$ and $dom(c') \subseteq vars(c(x)) \cup \bigcup_{i=1,\ldots,n} vars(c(v_i))$, from the fact that $h \in (\mathbf{x}, \mathbf{v})(\mathbf{v_1}, \mathbf{v}) \cdots (\mathbf{v_n}, \mathbf{v})$ we conclude that $c(v) \notin dom(c')$, $(cc')(v) = c(v)c' = c(v) \in V$ and $h' \in \mathbf{v}$.

Consider $vx(x, v) \Rightarrow v \in \alpha^V_{fr}(x = t(v_1, \ldots, v_n))$, with $t(v_1, \ldots, v_n) \notin \mathcal{V}$ and $n \geqslant 0$. Since $c(x) \in V$ we have $c' = \{c(x) = t(c(v_1), \ldots, c(v_n))\}$ and from $h \in v(\mathbf{x}, \mathbf{v})$ we conclude that $c(v) \neq c(x)$. Hence $(cc')(v) = c(v)c' = c(v) \in V$ and $h' \in \mathbf{v}$. □

Proof of Lemma 46. By Lemma 44 and Proposition 32 every $l \Rightarrow r \in \alpha^V_{aux}(c)$ is correct for c. We prove that if we substitute $(v_1, v_2) \in l$ with $v_1 v_2$, provided that $mgu(c(v_1), c(v_2))$ does not exist, the

resulting arrow $l' \Rightarrow r$ is correct for c. Let $h' = \exists_{W'} c' \in \mathbf{l}'$. If $vars(c'(v_1)) \cap vars(c'(v_2)) = \varnothing$ then $h' \in \mathbf{l}$ and $h' \bigstar^{H_V} c \in \mathbf{r}$ by the hypothesis that $c \in \mathbf{l} \to \mathbf{r}$. Otherwise, from $\{c'(v_1), c'(v_2)\} \subseteq V$ we conclude that $c'(v_1) = c'(v_2)$. Since $mgu(c(v_1), c(v_2))$ does not exist, we have that $h' \bigstar^{H_V} c$ does not exist too. □

Proof of Proposition 47. It is a corollary of Lemmas 44 and 46, by using Propositions 32, 35, and 40. □

Appendix D. Proofs of Section 10

Proof of Proposition 52. We prove that all the abstract operators of Section 8 are monotonic w.r.t. $\preceq$. This will entail the thesis by induction on c, since $\rho_V(A) \preceq A$ and $A_1 \preceq A_2$ entails $free_V(A_2) \subseteq free_V(A_1)$ and $indep_V(A_2) \subseteq indep_V(A_1)$.

Let $A, A_1, A_2 \in Rep_V$ be such that $A_1 \preceq A_2$.

We have $rename_{x \to n}^{Rep_V}(A_1) \preceq rename_{x \to n}^{Rep_V}(A_2)$.

For $restrict^{Rep_V}$, consider $l_2 \Rightarrow r \in restrict_x^{Rep_V}(A_2)$. Then $l_2 = l_2' \setminus X$ with $l_2' \Rightarrow r \in A_2$, $(x,x) \notin l_2'$ and X as defined in Definition 33. Then there exists $l_1' \Rightarrow r \in A_1$ such that $l_1' \subseteq l_2' \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2'\}$. Since $(x,x) \notin l_2'$, we have $(x,x) \notin l_1'$. Therefore, $l_1' \setminus X \Rightarrow r \in restrict_x^{Rep_V}(A_1)$, and $l_1' \setminus X \subseteq l_2' \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2'\} \setminus X = (l_2' \setminus X) \cup \{(v,v') \mid v, v' \in V \setminus x,\ v \neq v',\ (v,v) \in l_2' \setminus X\}$, since $(x,x) \notin l_2'$.

For $expand^{Rep_V}$, let $l_2 \Rightarrow r \in expand_n^{Rep_V}(A_2)$ with $l_2 \Rightarrow r \in A_2$. Then there exists $l_1 \Rightarrow r \in A_1$ with $l_1 \subseteq l_2 \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2\}$ and $l_1 \Rightarrow r \in expand_n^{Rep_V}(A_1)$. If $l_2[?_1 \mapsto n, ?_2 \mapsto ?_1] \Rightarrow r[?_1 \mapsto n, ?_2 \mapsto ?_1] \in expand_n^{Rep_V}(A_2)$ with $l_2 \Rightarrow r \in A_2$, then there exists $l_1 \Rightarrow r \in A_1$ with $l_1 \subseteq l_2 \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2\}$. Therefore, $l_1[?_1 \mapsto n, ?_2 \mapsto ?_1] \subseteq l_2[?_1 \mapsto n, ?_2 \mapsto ?_1] \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2\}[?_1 \mapsto n, ?_2 \mapsto ?_1] \subseteq l_2 7[?_1 \mapsto n, ?_2 \mapsto ?_1] \cup \{(v,v') \mid v, v' \in V \cup n,\ v \neq v', (v,v) \in l_2[?_1 \mapsto n, ?_2 \mapsto ?_1]\}$. Since $l_1[?_1 \mapsto n, ?_2 \mapsto ?_1] \Rightarrow r[?_1 \mapsto n, ?_2 \mapsto ?_1] \in expand_n^{Rep_V}(A_1)$, we have the thesis.

For $\cup^{Rep_V}$, consider $l_2 l \Rightarrow r \in \cup^{Rep_V}(A_2, A)$, with $l_2 \Rightarrow r \in A_2$ and $l \Rightarrow r \in A$. There exists $l_1 \Rightarrow r \in A_1$ with $l_1 \subseteq l_2 \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2\}$. Hence $l_1 l \subseteq l_2 l \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2\} \subseteq l_2 l \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_2 l\}$. Since $l_1 l \Rightarrow r \in \cup^{Rep_V}(A_1, A)$ and the same argument can be used for the symmetrical case of the definition of $\cup^{Rep_V}$, we have the thesis.

For $\bigstar^{Rep_V}$, consider $l_1 \cdots l_n \Rightarrow r \in A_2 \bigstar^{Rep_V} A$, with $r_1 \cdots r_n \Rightarrow r \in A \cup T$ (T is the set of Definition 30), $l_i \Rightarrow r_i' \in A_2 \cup T$ and $\mathbf{r}_\mathbf{i}' \subseteq \mathbf{r}_\mathbf{i}$ for $i = 1, \ldots, n$. Then $l_i' \Rightarrow r_i' \in A_1 \cup T$ with $l_i' \subseteq l_i \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_i\}$ and $l_1' \cdots l_n' \Rightarrow r \in A_1 \bigstar^{Rep_V} A$ with $l_1' \cdots l_n' \subseteq l_1 \cdots l_n \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_1 \cdots l_n\}$. Consider now $l_1 \cdots l_n \Rightarrow r \in A_2 \bigstar^{Rep_V} A$ with $r_1 \cdots r_n \Rightarrow r \in A_2 \cup T$, $l_i \Rightarrow r_i' \in A \cup T$ and $\mathbf{r}_\mathbf{i}' \subseteq \mathbf{r}_\mathbf{i}$. There exists $l' \Rightarrow r \in A_1 \cup T$ such that $l' \subseteq r_1 \cdots r_n \cup \{(v,v') \mid v, v' \in V,\ v \neq v', (v,v) \in r_1 \cdots r_n\}$. Given $k \in l'$, whether $k \in r_1 \cdots r_n$ or $k = (v,v')$ with $(v,v) \in r_1 \cdots r_n$. In both cases there exists i, $1 \leqslant i \leqslant n$, such that $\mathbf{r}_\mathbf{i}' \subseteq \mathbf{r}_\mathbf{i} \subseteq \mathbf{k}$, and we can select a set S of natural numbers between 1 and n such that $\bigcup_{i \in S} l_i \Rightarrow r \in A_1 \bigstar^{Rep_V} A$ and $\bigcup_{i \in S} l_i \subseteq l_1 \cdots l_n \subseteq l_1 \cdots l_n \cup \{(v,v') \mid v, v' \in V,\ v \neq v',\ (v,v) \in l_1 \cdots l_n\}$. The other case of $\bigstar^{Rep_V}$ is symmetrical. □

Proof of Proposition 53. Let $l \Rightarrow r, l' \Rightarrow r' \in Rep_V$. We write $l \Rightarrow r \preceq l' \Rightarrow r'$ if and only if $r = r'$ and $l \subseteq l' \cup \{(v, v') \mid v, v' \in V, v \neq v', (v, v) \in l'\}$. The relation $\preceq$ turns out to be a well founded partial order. We have $A \preceq A'$ if and only if for each arrow $a' \in A'$ there exists an arrow $a \in A$ such that $a \preceq a'$.

Given an arrow $a \in A$, let us consider the set of all the arrows $b \in A$ with $b \preceq a$, which we denote by $\downarrow a$. If $A' \preceq A$, the least element of $\downarrow a$, namely $\cap \downarrow a$, belongs to A'. It turns out that $\bar{A} = \{\cap(\downarrow a) \mid a \in A\}$ is the least subset of A according to the $\preceq$ ordering and that $\bar{A} = \rho^1(A)$.

We want to prove now that $\gamma_{Rep}(\rho^1(A)) = \gamma_{Rep}(A)$. It is enough to prove that, given two arrows a and a', if $a \preceq a'$ then $\gamma_{Rep}(a) \subseteq \gamma_{Rep}(a')$. If $a \preceq a'$, then $a = l \Rightarrow r$, $a' = l' \Rightarrow r$ and $\gamma_{Rep}(l) \supseteq \gamma_{Rep}(l')$. By contravariance we have $\gamma_{Rep}(a) \subseteq \gamma_{Rep}(a')$. □

Proof of Proposition 55. The map ρ^2_V, applied to $A \in Rep_V$, removes (v, v') from the left-hand side of $l \Rightarrow r$ if and only if $(v, v) \in l$ and $v \neq v'$. Therefore, it is reductive w.r.t. $\preceq$. Moreover, $dim(\rho^2_V(A)) \leqslant dim(A)$ and $\gamma_{Rep}(A) = \gamma_{Rep}(\rho^2_V(A))$. Indeed, given $h \in H_V$, if $h \in (v, v)$ then h is ground and $h \in (v, v')$ for any $v' \in V$. □

References

[1] G. Amato, F. Spoto, Abstract compilation for sharing analysis, in: H. Kuchen, K. Ueda (Eds.), Proceedings of FLOPS 2001, Lecture Notes in Computer Science, Tokyo, Japan, March, vol. 2024, Springer, Berlin, 2001, pp. 311–325.

[2] T. Armstrong, K. Marriott, P. Schachte, H. Søndergaard, Two classes of Boolean functions for dependency analysis, Science of Computer Programming 31 (1) (1998) 3–45.

[3] R. Bagnara, Data-flow analysis for constraint logic-based languages, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, 1997.

[4] R. Bagnara, P.M. Hill, E. Zaffanella, Set-Sharing is reduntant for pair-sharing, Theoretical Computer Science 277 (1 and 2) (2002) 3–46.

[5] A. Bossi, M. Gabbrielli, G. Levi, M. Martelli, The *s*-semantics approach: theory and applications, Journal of Logic Programming 19–20 (1994) 149–197.

[6] M. Bruynooghe, M. Codish, Freeness, sharing, linearity and correctness – all at once, in: P. Cousot, M. Falaschi, G. Filé, A. Rauzy (Eds.), Proceedings of the 3rd International Workshop on Static Analysis, Lectures Notes in Computer Science, vol. 724, Springer, Berlin, 1993, pp. 153–164.

[7] M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, A. Mulkers, A freeness and sharing analysis of logic programs based on a pre-interpretation, in: R. Cousot, D.A. Schmidt (Eds.), Proceedings of the 3rd International Symposium on Static Analysis, Lecture Notes in Computer Science, Aachen, Germany, vol. 1145, Springer, Berlin, 1996, pp. 128–142.

[8] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers 8 (35) (1986) 677–691.

[9] M. Codish, M. Bruynooghe, M.G. de la Banda, M. Hermenegildo, Exploiting goal independence in the analysis of logic programs, Journal of Logic Programming 32 (3) (1997) 247–261.

[10] M. Codish, D. Dams, G. Filé, M. Bruynooghe, Freeness analysis for logic programs – and correctness? in: D.S. Warren (Ed.), Proceedings of the 10th International Conference on Logic Programming, Budapest, Hungary, The MIT Press, Cambridge, MA, 1993, pp. 116–131.

[11] M. Codish, D. Dams, G. Filé, M. Bruynooghe, On the design of a correct freeness analysis for logic programs, Journal of Logic Programming 28 (3) (1996) 181–206.

[12] M. Codish, B. Demoen, Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop, in: Proceedings of the 1st International Symposium on Static Analysis, Lecture Notes in Computer Science, vol. 864, Springer, Berlin, 1994, pp. 281–296.

[13] M. Codish, V. Lagoon, F. Bueno, An algebraic approach to sharing analysis of logic programs, Journal of Logic Programming 42 (2) (2000) 110–149.
[14] A. Cortesi, G. Filè, W. Winsborough, Prop revisited: propositional formulas as abstract domain for groundness analysis, in: Proceedings of the 6th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Silver Spring, MD, 1991, pp. 322–327.
[15] A. Cortesi, G. Filè, W. Winsborough, Optimal groundness analysis using propositional logic, Journal of Logic Programming 27 (2) (1996) 137–167.
[16] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of the 6th ACM Symposium on Principles of Programming Languages, 1979, pp. 269–282.
[17] P. Cousot, R. Cousot, Abstract interpretation and applications to logic programs, Journal of Logic Programming 13 (2 and 3) (1992) 103–179.
[18] M.G. de la Banda, K. Marriott, P. Stuckey, H. Søndergaard, Differential methods in logic program analysis, Journal of Logic Programming 35 (1) (1998) 1–37.
[19] G. Filé, R. Giacobazzi, F. Ranzato, A unifying view on abstract domain design, ACM Computing Surveys 28 (2) (1996) 333–336.
[20] M. Gabbrielli, G. Levi, M.C. Meo, Observable behaviors and equivalences of logic programs, Information and Computation 122 (1) (1995) 1–29.
[21] M. Gabbrielli, M.C. Meo, Fixpoint semantics for partial computed answer substitutions and call patterns, in: H. Kirchner, G. Levi (Eds.), Proceedings of the 3rd International Conference on Algebraic and Logic Programming, Lecture Notes in Computer Science, vol. 632, Springer, Berlin, 1992, pp. 84–99.
[22] R. Giacobazzi, S.K. Debray, G. Levi, Generalized semantics and abstract interpretation for constraint logic programs, Journal of Logic Programming 25 (3) (1995) 191–247.
[23] R. Giacobazzi, F. Ranzato, Refining and compressing abstract domains, in: Proceedings of the ICALP'97 Conference, Lecture Notes in Computer Science, vol. 1256, Springer, Berlin, 1997, pp. 771–781.
[24] R. Giacobazzi, R. Ranzato, The reduced relative power operation on abstract domains, Theoretical Computer Science 216 (1999) 159–211.
[25] R. Giacobazzi, F. Scozzari, A logical model for relational abstract domains, ACM Transactions on Programming Languages and Systems 20 (5) (1998) 1067–1109.
[26] M. Hermenegildo, W. Warren, S.K. Debray, Global flow analysis as a practical compilation tool, Journal of Logic Programming 13 (2 and 3) (1992) 349–366.
[27] P.M. Hill, F. Spoto, Freeness analysis through linear refinement, in: Proceedings of the 6th International Symposium on Static Analysis, Venice, Italy, Lecture Notes in Computer Science, Springer, Berlin, 1999, pp. 85–100.
[28] D. Jacobs, A. Langen, Static analysis of logic programs for independent And-parallelism, Journal of Logic Programming 13 (2 & 3) (1992) 291–314.
[29] A. King, A synergistic analysis for sharing and groundness which traces linearity, in: D. Sannella (Ed.), Proceedings of the 5th European Symposium on Programming, Lecture Notes in Computer Science, Edinburgh, UK, vol. 788, Springer, Berlin, 1994, pp. 363–378.
[30] A. King, P. Soper, Depth-k sharing and freeness, in: P. van Hentenryck (Ed.), Proceedings of the 11th International Conference on Logic Programming, Santa Margherita Ligure, Italy, 1994, pp. 553–568.
[31] A. Langen, Advanced techniques for approximating variable aliasing in logic programs, Ph.D. Thesis, University of Southern California, 1991.
[32] G. Levi, F. Spoto, An experiment in domain refinement: type domains and type representations for logic programs, in: C. Palamidessi, H. Glaser, K. Meinke (Eds.), Principles of Declarative Programming, Lecture Notes in Computer Science, Pisa, Italy, September, vol. 1490, Springer, Berlin, 1998, pp. 152–169.
[33] G. Levi, F. Spoto, Non pair-sharing and freeness analysis through linear refinement, in: Proc. of the Partial Evaluation and Program Manipulation Workshop, Boston, MA, January 2000, ACM Press, pp. 52–61. Available from http://www.sci.univr.it/~spoto/papers.html.
[34] K. Marriott, H. Søndergaard, Precise and efficient groundness analysis for logic programs, ACM letters on Programming Languages and Systems 2 (1–4) (1993) 181–196.
[35] A. Martelli, U. Montanari, An efficient unification algorithm, ACM Transactions on Programming Languages and Systems 4 (1982) 258–282.

[36] K. Muthukumar, M. Hermenegildo, Combined determination of sharing and freeness of program variables through abstract interpretation, in: K. Furukawa (Ed.), Proceedings of the 8th International Conference on Logic Programming, Paris, The MIT Press, Cambridge, MA, 1991, pp. 49–63.
[37] F. Scozzari, Logical optimality of groundness analysis, Theoretical Computer Science 277 (1 and 2) (2002) 149–184.
[38] H. Søndergaard, An application of abstract interpretation of logic programs: occur check reduction, in: B. Robinet, R. Wilhelm (Eds.), Proceedings of the European Symposium on Programming, Lecture Notes in Computer Science, vol. 213, Springer, Berlin, 1986, pp. 327–338.